Software Engineering Institute
Carnegie Mellon University

# A FRAMEWORK FOR SOFTWARE PRODUCT LINE PRACTICE, VERSION 5.0

*Linda M. Northrop and Paul C. Clements*
*With Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, John McGregor, and Liam O'Brien*

December 2012

## What's New in Version 5.0

A *Framework for Software Product Line Practice* is a document that aids the soft-ware community in software product line endeavors. Each version represents an incremental attempt to capture the latest information about successful software product line practices. This information has been gleaned from studies of organizations that have built product lines, from direct collaborations on software product lines with customer organizations, and from leading practitioners in software product lines.

In addition to this framework, other resources are available from the Software Engineering Institute (SEI), including

- the book *Software Product Lines: Practices and Patterns* [Clements 2002c], that contains a previous version of this framework, plus three comprehensive case studies of product line organizations and a rich set of product line practice patterns to aid in the adoption of software product line practice
- the SEI's product line practice Web site, where you can download the latest publications and more product line case studies and learn about upcoming events
- a five-course Software Product Line Curriculum, with courses based on extensive SEI and community experience in developing, acquiring, and fielding software product lines. The curriculum equips software professionals with state-of-the-art practices, so they can efficiently use proven product line practices to achieve their strategic reuse and other business goals.

There are significant changes in Version 5.0 that reflect current prevalent trends in software engineering (the open source movement, globally distributed development, service-oriented architectures, model-driven development, and agile development), as well as a wave of new product line experiences that have surfaced new practices and references. The resultant changes to the framework include

- a discussion of contextual factors that influence core asset development. Product constraints, production constraints, production strategy, and preexisting assets are no longer rigid "inputs" but rather contextual factors.
- more in-depth coverage of the production plan concept
- new FAQs
- significant modifications to the following practice areas:

– "COTS Utilization" was changed to "Using Externally Available Software" and expanded to include open source software and services as prominent software choices.
– "Mining Existing Assets" was expanded to include externally available software as a mining source.
– "Software System Integration" was expanded to discuss continuous, iterative integration, and closer ties have been made to production planning.
– "Testing" now includes more detailed descriptions of guidelines and test artifacts.
– "Data Collection, Measurement, and Tracking" was changed to "Measurement and Tracking" and rewritten to emphasize the goals of measurement rather than the mechanics.
– "Process Definition" was changed to "Process Discipline" and rewritten to emphasize the need for process discipline throughout, which exceeds the ability to merely define processes. This practice area was also expanded to include a treatment of agile approaches and the production process and production method for the product line.
– "Technical Planning" was expanded to provide more coverage of production planning.
– "Tool Support" was expanded to discuss tools for automating product derivation.
– "Building a Business Case" now includes an expanded discussion of using the business case to weigh alternative strategies and coverage of Boehm's COPLIMO economic model for product lines.
– "Customer Interface Management" now includes a discussion of customer identification.
– "Developing an Acquisition Strategy" now covers the acquisition of services and other externally available software, as well as acquisition in a global development environment.
– "Launching and Institutionalizing" now includes a discussion of the SEI Adoption Factory pattern and more coverage of institutionalization.
– "Operations" now considers a product line concept of operations (CONOPS) to be essential.
– "Structuring the Organization" now has a clarified relationship with the "Operations" practice area.
– "Training" includes added sources of product line training.

Future versions will build on the current foundation by growing the body of knowledge and refining what is already there.

## Introduction

A product line is a set of products that together address a particular market segment or fulfill a particular mission. Product lines are, of course, nothing new in manufacturing. Airbus builds one, and so do Ford, Dell, and even McDonald's. Each of these companies exploits commonality in different ways. Boeing, for example, developed the 757 and 767 transports in tandem, and the parts lists for these very two different aircraft overlap by about 60%, achieving significant economies of production and maintenance. But *software* product lines based on interproduct commonality are a relatively new concept that is rapidly emerging as a viable and important software development paradigm. Product flexibility is the anthem of the software marketplace, and product lines fulfill the promise of tailor-made systems built specifically for the needs of particular customers or customer groups. A product line succeeds because the commonalities shared by the software products can be exploited to achieve economies of production. The products are built from common assets in a prescribed way.

Companies are finding that this practice of building sets of related systems from common assets can yield remarkable quantitative improvements in productivity, time to market, product quality, and customer satisfaction. They are finding that a software product line can efficiently satisfy the current hunger for mass customization. Organizations that acquire, as opposed to build, software systems are finding that commissioning a set of related systems as a commonly developed product line yields economies in delivery time, cost, simplified training, and streamlined acquisition.

But along with the gains come risks. Using a product line approach constitutes a new technical strategy for the organization. Organizational and management issues constitute obstacles that are critical to overcome and often add more risk, because they are less obvious. Building a software product line and bringing it to market requires a blend of skillful engineering as well as both technical and organizational management. Acquiring a software product line also requires this same blend of skills to position the using organizations, so they can effectively exploit the commonality of the incoming products, as well as lend sound technical oversight and monitoring to the development effort. These skills are necessary to overcome the pitfalls that may bring failure to an unsophisticated organization.

We've worked to gather information and identify key people with product line experience. Through surveys, workshops, conferences, case studies, and direct collaboration with organizations on product line efforts, we have amassed and categorized a reservoir of information. Organizations that have succeeded with product lines vary widely in

- the nature of their products
- their market or mission
- their business goals
- their organizational structure
- their culture and policies
- their software process discipline
- the maturity and extent of their legacy artifacts

Nevertheless, there are universal essential activities and practices that emerge, having to do with the ability to construct new products from a set of common assets while working under the constraints of various organizational contexts and starting points. This document describes a *framework*[1] for product

1. Our use of the word *framework* is meant to suggest a conceptual index–a frame of reference–for the information essential to success with software product lines. We are using the dictionary definition with no intended connections to current technical usages in the vein of architectural or application frameworks.

line development. The framework is a product line encyclopedia; it is a Web-based document describing the essential activities and practices, in both the technical and organizational areas. These activities and practices are those in which an organization must be competent before it can reap the maximum benefit from fielding a product line of software or software-intensive systems. The audience for this framework includes members of an organization who are in a position to make or influence decisions regarding the adoption of product line practices, as well as those who are already involved in a product line effort.

## Purpose

The goals of this framework are

- **to identify the foundational concepts underlying software product lines and the essential activities to consider before developing a product line**
- **to identify practice areas that an organization developing software product lines must master**
  Although these practice areas may be required for engineering any software system, the product line context imposes special constraints requiring that they be carried out in an unconventional way.
- **to define example practices in each practice area, where current knowledge is sufficient to do so**
  For example, "Configuration Management" is a practice area that applies to any software development effort, but it has special implications for product line development. Thus, we identify it as a practice area and define one or more effective configuration management practices for product lines. In many cases, the definition of the practice is a reference to a source outside this document.
- **to provide guidance to an organization about how to move to a product line approach for software**
  Organizations using this framework should be able to understand the state of their product line capabilities by (1) understanding the set of essential practice areas, (2) assessing how practices in those areas differ from their conventional forms for single product development, and (3) comparing that set of practices to the organization's existing skill set.

As such, this framework can serve as the basis for a technology and improvement plan aimed at achieving product line development goals.

Every organization is different and comes to the product line approach with different goals, missions, assets, and requirements. Practices for a product line builder will be different from those for a product line acquirer, and different still for a component or service vendor. Appropriate practices will also vary according to

- the type of system being built
- the depth of the organization's domain experience
- the legacy assets on hand
- the organizational goals

- the maturity of artifacts and processes
- the skill level of the personnel available
- the production strategy embraced
- many other factors

There is no one correct set of practices for every organization; hence, we do not prescribe a methodology consisting of a set of specific practices. The framework is not a maturity model[2] or a process guide. We are prescriptive about the practice areas and prescribe that organizations adopt appropriate practices in each practice area. This document contains practices that we have seen work successfully.

This framework has been used by organizations, large and small, to help them plan to adopt a product line approach, as well as to help them gauge how they're doing and in what areas they're falling short. We use it to guide our collaborations with customers and to focus in on the areas where our collaboration will best assist our customers. We also use it as the basis for conducting an SEI Product Line Technical Probe (PLTP)—formal diagnostics of an organization's product line fitness [Clements 2002c, Ch. 8; SEI 2007e]. The framework is a living, growing document that represents our best picture of sound product line practice as described to us by its many users and reviewers—all of whom are practitioners. The framework has served successfully as the basis for technology and improvement plans that achieve product line goals.

## What Is a Software Product Line?

A *software product line* is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

This definition is consistent with the definition traditionally given for any product line. But it adds more: it puts constraints on the way in which the systems in a software product line are developed. Why? Because substantial production economies can be achieved when the systems in a software product line are developed from a common set of assets in a prescribed way, in contrast to being developed separately, from scratch, or in an arbitrary fashion. It is exactly these production economies that make the software product line approach attractive.

How is production made more economical? Each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance, adding any new components that may be necessary, and assembling the collection according to the rules of a common, product-line-wide architecture. Building a new product (system) becomes more a matter of assembly or generation than one of creation; the predominant activity is integration rather than programming. For each software product line, there is a predefined guide or plan that specifies the exact product-building approach.

[2] For a discussion of the relationship between Capability Maturity Model Integration (CMMI) and this framework, see the work of Jones and Soule [Jones 2002a].

Certainly the desire for production economies is not a new business goal, and neither is a product line solution. But a *software product line* is a relatively new idea, and it should seem clear from our description that software product lines require a different technical tack. The more subtle consequence is that software product lines require much more than new technical practices.

The common set of assets and the plan for how they are used to build products don't just materialize without planning, and they certainly don't come free. They require organizational foresight, investment, planning, and direction. They require strategic thinking that looks beyond a single product. The disciplined use of the common assets to build products doesn't just happen either. Management must direct, track, and enforce the use of the assets. Software product lines are as much about business practices as they are about technical practices.

Software product lines give *economies of scope*, which means that you take economic advantage of the fact that many of your products are very similar—not by accident, but because you planned it that way. You make deliberate, strategic decisions and are systematic in effecting those decisions.

## What Software Product Lines Are Not

Many approaches can, at first blush, be confused with software product lines. Describing what you don't mean is often as instructive as describing what you do mean. When we speak of software product lines, we don't mean

- fortuitous, small-grained reuse
- single-system development with reuse
- just component-based or service-based development
- just a reconfigurable architecture
- releases and versions of single products
- just a set of technical standards

These approaches are described further below.

### Fortuitous, Small-Grained Reuse

Reuse, as a software strategy for decreasing development costs and improving quality, is not a new idea, and software product lines definitely involve reuse—reuse, in fact, of the highest order. So what's the difference? Many past reuse agendas have focused on the reuse of relatively small pieces of code—that is, small-grained reuse. Organizations have built reuse libraries containing algorithms, modules, objects, or components. Almost anything a software developer writes goes into the library. Other developers are then urged (and sometimes required) to use what the library provides instead of creating their own versions. Unfortunately, it often takes longer to locate these small pieces and integrate them into a system than it would take to build them anew. Documentation, if it exists at all, might explain the situation for which the piece was created but not how it can be generalized or adapted to other situations. The benefits of small-grained reuse depend on the predisposition of the software engineer to use what is in the library, the suitability of what is in the library for the engineer's particular needs, and the successful adaptation and integration of the library units into the rest of the system. If reuse occurs at all under these conditions, it is fortuitous, and the payoff is usually nonexistent.

In a software product line approach, the reuse is planned, enabled, and enforced—the opposite of opportunistic. The asset base includes those artifacts in software development that are most costly to develop from scratch—namely, the requirements, domain models, software architecture, performance models, test cases, and components. All of the assets are designed to be reused and are optimized for use in more than a single system. The reuse with software product lines is comprehensive, planned, and profitable.

### Single-System Development with Reuse

Suppose you are developing a new system that seems very similar to one you have built before. You borrow what you can from your previous effort, modify it as necessary, add whatever it takes, and field the product, which then assumes its own maintenance trajectory separate from that of the first product. What you have done is what is called "clone and own." You certainly have taken economic advantage of previous work; you have reused a part of another system. But now you have two entirely different systems, not two systems built from the same base. This is again ad hoc reuse.

There are two major differences between this approach and a software product line approach. First, software product lines reuse assets that were designed explicitly for reuse. Second, the product line is treated as a whole, not as multiple products that are viewed and maintained separately. In mature product line organizations, the concept of multiple products disappears. Each product is simply a tailoring of the common assets that constitute the core of each product, plus perhaps a small collection of additional artifacts unique to that product. It is the core assets that are designed carefully and evolved over time. It is the core assets that are the organization's premiere intellectual property.

### Just Component-Based or Service-Based Development

Software product lines rely on a form of component-based (or service-based) development, but much more is involved. The typical definition of such development involves either the selection of in-house components (services) from in-house or the marketplace to build products. Although the products in software product lines certainly are composed of components—some of which may be Web services—these components are all specified by the product line architecture. Moreover, the components are assembled in a prescribed way, which includes exercising built-in variation mechanisms in the components to put them to use in specific products. The prescription comes from both the architecture and the production plan and is missing from standard component-based development. In a product line, the generic form of the component is evolved and maintained in the core asset base. In component-based development, if any variation is involved, it is usually accomplished by writing code, and the variants are most likely maintained separately. Component-based development also lacks the technical and organizational management aspects that are so important to the success of a software product.

### Just a Reconfigurable Architecture

Reference architectures and application frameworks are designed to be reused in multiple systems and to be reconfigured as necessary. Reusing architectural structures is a good idea because the architecture is a pivotal part of any system and a costly one to construct. A product line architecture is designed to support the variation needed by the products in the product line, so making it reconfigurable makes sense. However, the product line architecture is just one asset, albeit an important one, in the product line's core asset base.

### Releases and Versions of Single Products

Organizations routinely produce new releases and versions of products. Each of these new versions and releases is typically constructed using the architecture, components, test plans, and other features of the prior releases. Why are software product lines different? First, a product line has multiple simultaneous products, all of which are going through their own cycles of release and versioning simultaneously. Thus, the evolution of a single product must be considered within a broader context—namely, the evolution of the product line as a whole. Second, in a single-product context, once a product is updated, there's often no looking back—whatever went into the production of earlier products is no longer considered to be of any value, or at best, is retired as soon as practicable. But in a product line, an early version of a product that is still considered to have market potential can easily be kept as a viable member of the family: it is, after all, an instantiation of the core assets, just like other versions of other products.

### Just a Set of Technical Standards

Many organizations set up technical standards to limit the choices their software engineers can make regarding the kinds and sources of components to incorporate in systems. They audit for compliance at architecture and design reviews to ensure that the standards are being followed. For example, the developer might be able to select between three identified database choices and two identified Web browsers but must use a specific middleware or spreadsheet product, if either is necessary. Technical standards are constraints to promote interoperability and decrease the cost associated with the maintenance and support of commercial components. An organization that undertakes a product line effort may have such technical standards, in which case the product line architecture and components will need to conform to those standards. However, the standards are simply constraints that are input to the software product line, no more.

## Benefits and Costs of a Product Line

Software product line approaches accrue benefits at multiple levels. This section lists the benefits (and some of the costs) from the perspective of the organization as a whole, the individuals within the organization, and the core assets involved in software product line production.

## Organizational Benefits

The organizations that we have studied[3] have achieved remarkable benefits that are aligned with commonly held business goals including

- large-scale productivity gains
- decreased time to market
- increased product quality
- decreased product risk
- increased market agility
- increased customer satisfaction
- more efficient use of human resources
- ability to effect mass customization
- ability to maintain market presence
- ability to sustain unprecedented growth

These benefits give organizations a competitive advantage and are derived from the reuse of the core assets in a strategic and prescribed way. Once the core asset base for the product line is established, there is a direct savings *each* time a product is built. That savings is associated with *each* of the following:

- **requirements:** There are common product line requirements. Product requirements are deltas to this established requirements base. Extensive requirements analysis is saved, and feasibility is assured.
- **architecture:** An architecture for a software system represents a large investment of time from the organization's most talented engineers. The quality goals for a system—its performance, reliability, modifiability, and so on—are largely allowed or precluded once the architecture is in place. If the architecture is wrong, the system cannot be saved. The product line architecture is used for each product and need only be instantiated. Considerable time and risk are spared.
- **components:** Up to 100% of the components in the core asset base are used in each product. These components may need to be altered using inheritance or parameters, but the design is intact, as are data structures and algorithms. In addition, the product line architecture provides component specifications for all the non-unique components that may be necessary.
- **modeling and analysis:** Performance models and the associated analyses are existing product line core assets. With each new product, there is extremely high confidence that the timing problems have been worked out and the bugs associated with distributed computing—synchronization, network loading, and absence of deadlock—have been eliminated.

---

[3] The SEI has published several detailed case studies of successful product line organizations and the benefits they enjoyed. You can find these case studies in the book Software Product Lines: Practices and Patterns [Clements 2002c] and on the Web at http://www.sei.cmu.edu/library/abstracts/books/0201703327.cfm. You can also find references to product line efforts at http://splc.net/fame.html and http://www.sei.cmu.edu/productlines/casestudies/.

- **testing:** Generic test plans, test processes, test cases, test data, test harnesses, and the communication paths required to report and fix problems have already been built. These testing artifacts need only be tailored on the basis of the variations related to the product.
- **planning:** The production plan has already been established. Baseline budgets and schedules from previous product development projects already exist and provide a reliable basis for the product work plans.
- **processes:** Configuration control boards, configuration management tools and procedures, management processes, and the overall software development process are in place, have been used before, and are robust, reliable, and responsive to the organization's special needs.
- **people:** Fewer people are required to build products, and the people are more easily transferred across the entire line.

A software product line approach provides options to future market opportunities. Even though exact opportunities and their certainty are impossible to predict, by exercising variation points in the core assets, product lines permit low-cost, low-risk experiments to explore opportunities.

Product lines enhance quality. Each new system takes advantage of all the defect elimination in its forebears; developer and customer confidence both rise with each new instantiation. The more complicated the system, the higher the payoff for solving the vexing performance, distribution, reliability, and other engineering issues once for the entire family.

## Individual Benefits

The benefits to individuals within an organization depend on their respective roles. Table 1 shows observed benefits for some of the individual stakeholders in the product line organization.

*Table 1: Product Line Benefits for Individual Stakeholders*

| Stakeholder Role | Benefits |
|---|---|
| Chief executive officer (CEO) | Options to quickly develop new products; large productivity gains; greatly improved time to market; sustained growth and market presence; the options and ability to economically capture a market niche |
| Chief operating officer (COO) | Efficient use of workforce; ability to explore new markets, new technology, and/or new products; fluid personnel pool |
| Technical manager | Increased predictability; well-established roles and responsibilities; efficient production |
| Software product developer | Higher morale; greater job satisfaction; can focus on truly unique aspects of products; easier software integration; fewer schedule delays; greater mobility within the organization; more marketable; have time to learn new technology; are part of a team building products with an established quality record and reputation |
| Architect or core asset developer | Greater challenge; work has more impact; prestige within the organization; become as marketable as the product line |
| Marketer | Predictable high-quality products; predictable delivery; can sell products with a pedigree |
| Customer | High-quality products; predictable delivery date; predictable cost; known costs for unique requirements; well-tested training materials and documentation; shared maintenance costs; potential to participate in a user's group |
| End user | Fewer defects; better training materials and documentation; a network of other users |

## Benefits Versus Costs

We have established that the strategic reuse of core assets that defines product line practice represents an opportunity for benefits across the board, but the picture is not yet complete. Launching a software product line is a business decision that should not be made randomly. Any organization that launches a product line should have in mind specific and solid business goals that it plans to achieve through product line practice. Moreover, the benefits given above should align carefully with the achievement of those goals, because a software product line requires a start-up investment as well as ongoing costs to maintain the core assets. We have already listed the benefits associated with the reuse of particular core assets. Usually a cost and a caveat are associated with the achievement of each benefit. Table 2 gives a partial list of core assets with the typical additional costs. We repeat the benefits for the sake of comparison.

*Table 2: Costs and Benefits of Product Lines*

| Core Asset | Benefit | Additional Costs |
|---|---|---|
| **Requirements:** The requirements are written for the group of systems as a whole, with requirements for individual systems specified by a delta or an increment to the generic set. | Commonality and variation are documented explicitly, which will help lead to an architecture for the product line. New systems in the product line will be much simpler to specify, because the requirements are reused and tailored. | Capturing requirements for a group of systems may require sophisticated analysis and intense negotiation to agree on both common requirements and variation points that are acceptable for all the systems. |
| **Architecture:** The architecture for the product line is the blueprint for how each product is assembled from the components in the core asset base. | Architecture represents a significant investment by the organization's most talented engineers. Leveraging this investment across all products in the product line means that for subsequent products, the most important design step is largely completed. | The architecture must support the variation inherent in the product line, which imposes an additional constraint on the architecture and requires greater talent to define. |
| **Software components:** The software components that populate the core asset base form the building blocks for each product in the product line. Some will be reused without alteration. Others will be tailored according to prespecified variation mechanisms. | The interfaces for components are reused. For actual components that are reused, the design decisions, data structures, algorithms, documentation, reviews, code, and debugging effort can all be leveraged across multiple products in the product line. | The components must be designed to be robust and extensible so that they are applicable across a range of product contexts. Variation points must be built in or at least anticipated. Often, components must be designed to be more general without loss of performance. |
| **Performance modeling and analysis:** For products that must meet real-time constraints (and some that have soft real-time constraints), analysis must be performed to show that the system's performance will be adequate. | A new product can be fielded with high confidence that real-time and distributed-systems problems have already been worked out, because the analysis and modeling can be reused from product to product. Process scheduling, network traffic loads, deadlock elimination, data consistency problems, and the like will all have been modeled and analyzed. | Reusing the analysis may impose constraints on moving processes among processors, creating new processes, or synchronizing existing processes. |

| Core Asset | Benefit | Additional Costs |
|---|---|---|
| **Business case, market analysis, marketing collateral, and cost and schedule estimates:** These are the up-front business necessities involved in any product. Generic versions are built that support the entire product line. | All the business and management artifacts involved in turning out the product line already exist at least in a generic form and can be reused. | All these artifacts must be generic or be made extensible to accommodate product variations. |
| **Tools and processes for software development and making changes:** The infrastructure for turning out a software product requires specific product line processes and appropriate tool support. | Configuration control boards, configuration management tools and procedures, management processes, and the overall software development process are all in place and have been used before. Tools and environments purchased for one product can be amortized across the entire product line. | The boards, process definitions, tools, and procedures must be more robust to account for unique product line needs and the differences between managing a product line and managing a single product. |
| **Test cases, test plans, and test data:** There are generic testing artifacts for the entire set of products in the product line with variation points to accommodate product variation. | Test plans, test cases, test scripts, and test data have already been developed and reviewed for the components that are reused. Testing artifacts represent a substantial organizational investment. Any saving in this area is a benefit. | All the testing artifacts must be more robust, because they will support more than one product. In addition, they must be extensible to accommodate variation among the products. |
| **People, skills, and training:** In a product line organization, even though members of the development staff may work on a single product at a time, they are, in reality, working on the entire product line. The product line is a single entity that embraces multiple products. | Because of the commonality of the products and the production process, personnel can be more easily transferred among product projects as required. Staff expertise is usually applicable across the entire product line. Their productivity, when measured by the number of products to which their work applies, rises dramatically. Resources spent on training developers to use processes, tools, and system components are expended only once. | Personnel must be trained beyond general software engineering and corporate procedures to ensure that they understand software product line practices and can use the core assets and procedures associated with the product line. New personnel must be much more specifically trained for the product line. Training materials must be created that address the product line. As product lines mature, the skills required in an organization tend to change, away from programming and toward relevant domain expertise and technology forecasting. This transition must be managed. |

For each of these core assets, the investment cost is usually much less than the value of the benefit. Also, most of the costs are up-front costs associated with establishing the product line. The benefits, on the other hand, accrue with each new product release. Once the approach is established, the organization’s productivity accelerates rapidly, and the benefits far outweigh the costs. However, an organization that attempts to institute a product line without being aware of the costs is likely to abandon the product line concept before seeing it through.

It takes a certain degree of maturity in the developing organization to field a product line successfully. Technology change is not the only barrier to successful product line adoption. Changes in management and organizational practices are also involved. Successful adoption of software product line practice is a careful blend of technological, process, organizational, and business improvements.

Organizations of all stripes have enjoyed quantitative benefits from their product lines. Product line practitioners have also shared with us examples of the costs, such as

- canceling three large projects so that sufficient resources could be devoted to the up-front development of core assets

- reassigning staff who could not adjust to the product line way of doing business
- suspending product delivery for an extended period of time while putting the new practices into place

Certainly not every organization must undergo such dramatic measures to adopt the software product line approach. And the companies that bore these costs and made the successful transition to product line practice all agree that the payoff more than compensated for the effort. However, these costs underscore the point that product line practice is often uncharted territory and may not be the right path for every organization.

## A Note on Terminology

In this document we use the following terms:

- A *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. (This is the definition we provided in "What Is a Software Product Line?")
- *Core assets* are those reusable artifacts and resources that form the basis for the software product line. Core assets often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation, specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions. The architecture is key among the collection of core assets.
- *Development* is a generic term used to describe how core assets (or products) come to fruition. Software enters an organization in any one of three ways: (1) the organization can build it itself (either from scratch or by mining legacy software), (2) purchase it or license it (using it largely unchanged), or (3) commission it (contract with someone else to develop it especially for that organization). So our use of the term *development* may actually involve building, acquisition, purchase, retrofitting earlier work, or any combination of these options. We recognize and address these options, but we use *development* as the general term.
- A *domain* is a specialized body of knowledge, an area of expertise, or a collection of related functionality. For example, the telecommunications domain is a set of telecommunications functionality, which, in turn, consists of other domains such as switching, protocols, telephony, and networks. A telecommunications software product line is a specific set of software systems that provide some of that functionality.
- *Software product line practice* is the systematic use of core assets to assemble, instantiate, or generate the multiple products that constitute a software product line. The appropriate verb depends on the production method for the product line. Software product line practice involves strategic, large-grained reuse.

Some practitioners use a different set of terms to convey essentially the same meaning. For some, a *product line* is a profit and loss center concerned with turning out a set of products; it refers to a business unit, not a set of products. The *product family* is that set of products we call the product line. The software assets in the core asset base are sometimes called a *platform*. What we call core asset development is sometimes referred to as *domain engineering*, and what we call product development is sometimes referred to as *application engineering*.

The terminology is not as important as the concepts. That having been said, you might encounter different terms in other places and should be able to translate them. These and other related terms are defined in the glossary.

### Starting Versus Running a Product Line

Many of the practice areas in this framework are written from the point of view of describing an in-place product line capability. We recognize that the framework will be used to help an organization put that capability in place, and ramping up to a product line is, in many ways, different than running one on a day-to-day basis.

We felt it was important to describe the end or “steady state” so that readers could understand the goals of the product line approach. However, to address the issues of starting (rather than running) a product line effort, see the “Launching and Institutionalizing” practice area and the SEI Adoption Factory pattern [Northrop 2004a].

## Product Line Essential Activities

At its essence, fielding a product line involves core asset development and product development using the core assets, both under the aegis of technical and organizational management. Core asset development and product development from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products. Often, products and core assets are built in concert with each other. Figure 1 illustrates this triad of essential activities.

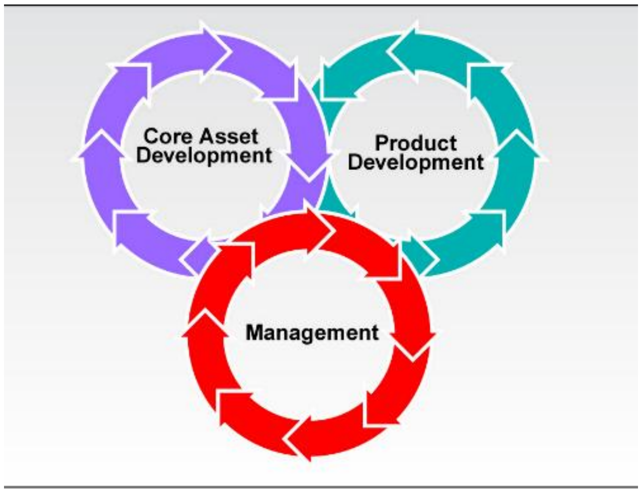


*Figure 1: The Three Essential Activities for Software Product Lines*

Each rotating circle represents one of the essential activities. All three are linked together and in perpetual motion, showing that they are all essential, inextricably linked, and highly iterative, and can occur in any order.

The rotating arrows indicate not only that core assets are used to develop products, but also that revisions of existing core assets or even new core assets might, and most often do, evolve out of product development. Figure 1 is neutral in regard to which part of the effort is launched first. In some contexts, already existing products are mined for generic assets—perhaps a requirements specification, an architecture, or software components—which are then migrated into the product line's core asset base. In other cases, the core assets may be developed or procured for later use in the production of products.

There is a strong feedback loop between the core assets and the products. Core assets are refreshed as new products are developed. The use of core assets is tracked, and the results are fed back to the core asset development activity. In addition, the value of the core assets is realized through the products that are developed from them. As a result, the core assets are made more generic by considering potential new products on the horizon. There is a constant need for strong, visionary management to invest resources in the development and sustainment of the core assets. Management must also precipitate the cultural change to view new products in the context of the available core assets. Either new products must align with the existing core assets, or the core assets must be updated to reflect the new products that are being marketed. Iteration is inherent in product line activities—that is, in turning out core assets and products and in coordinating the two. In the next three sections, we examine the three essential activities in greater detail.

## Core Asset Development

The goal of the core asset development activity is to establish a production capability for products. Figure 2 illustrates the core asset development activity along with its outputs and influential contextual factors.

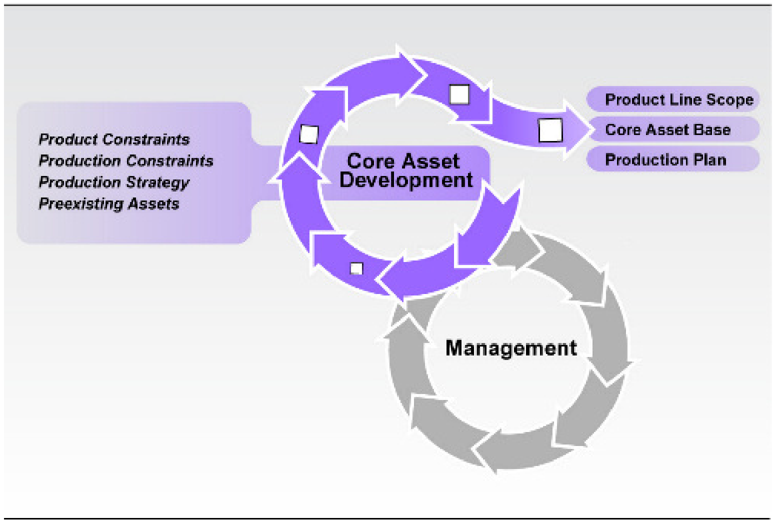


*Figure 2: Core Asset Development*

This activity, like its counterparts, is iterative. The rotating arrows suggest that there is no one-way causal relationship from the context to outputs; the activity of producing core assets might change the context. For example, expanding the product line scope (one of the outputs) may admit whole new classes of systems to examine as possible sources of legacy assets (part of the context). Similarly, a production constraint (such as the need to rapidly assemble products) may lead to restrictions on the product line architecture (an output). This restriction, in turn, will determine which preexisting assets (another contextual factor) are candidates for reuse or mining.

Core asset development does not happen in a vacuum; rather, it happens within a situational context of existing constraints and resources. This context influences how core asset development is carried out and the nature of the outputs it produces. Four of the most important contextual factors are

1. **product constraints:** What commonalities and variations exist among the products that will constitute the product line? What behavioral features do they provide? What features do the market and technology forecasts say will be beneficial in the future? What commercial, military, or company-specific standards apply to the products? What performance limits must they observe? With what external systems must they interface? What physical constraints must be observed? What quality requirements (such as availability and security) are imposed? The core assets must capitalize on the commonalities and accommodate envisioned variation with minimal tradeoff to product quality drivers such as security, reliability, usability, and so forth. These constraints may be derived from a set of preexisting products that will form the basis for the product line, they may be generated anew, or some combination of the two.

   The "Requirements Engineering," "Scoping," and "Understanding Relevant Domains" practice areas, which are described later, are concerned with gathering the relevant product constraints. These constraints affect the design of the core assets during, for example, architecture definition and component development.

2. **production constraints:** Must a new product be brought to market in a year, a month, or a day? What production capability must be given to engineers in the field? What company-specific

standards for software development processes must be followed in during product production? Who will be building the products and what environments will they use? Answering these and similar questions will drive decisions about, for example, whether to invest in a generator environment or rely on manual coding. These answers, in turn, will drive decisions about what kind of variation mechanisms to provide in the core assets and what production process will be used for products and eventually codified in the production plan.

Architecture definition and component development reflect the production constraints in the variation mechanisms the designers choose. And make/buy/mine/commission analysis is carried out under the influence of those constraints. The “Process Discipline” and “Technical Planning” practice areas, which are described later, are concerned with how production constraints are addressed in the production plan.

3. **production strategy:** The production strategy is the overall approach for realizing both the core assets and products. Will the product line be built proactively (starting with a set of core assets and spinning products off of them), reactively (starting with a set of products and generalizing their components to produce the product line core assets), or using some combination of the two (see “All Three Together”)? What will the transfer pricing strategy be—that is, how will the cost of producing the generic components be divided among the cost centers for the products? Will generic components be produced internally or purchased on the open market? How will the production of core assets be managed? The production strategy dictates the genesis of the architecture and its associated components and the path for their growth. Informed by the product constraints and the production constraints, the production strategy also drives the process by which products will be developed from core assets. Will products be generated automatically from the core assets, or will they be assembled?

   The production strategy is most closely tied to the “Funding,” “Launching and Institutionalizing,” “Market Analysis,” “Building a Business Case,” “Process Discipline,” “Technical Planning,” “Scoping,” “Architecture Definition,” and “Make/Buy/Mine/Commission Analysis” practice areas.

4. **preexisting assets:** Legacy systems and existing products embody an organization’s domain expertise and/or define its market presence. The product line architecture, or at least pieces of it, may borrow heavily from proven designs of related legacy systems or existing products. Components may be mined from legacy systems. Such components may represent key intellectual property of the organization in relevant domains and therefore become prime candidates for components in the core asset base. What software and organizational assets are available at the outset of the product line effort? Are there libraries, frameworks, algorithms, tools, components, and services that can be used? Are there technical management processes, funding models, and training resources that can be adapted easily for the product line? Through careful analysis, an organization determines what is most appropriate to use. However, preexisting assets are not limited to assets that were built by the product line organization. Externally available software such as COTS, Web services, and open source products, as well as standards, patterns, and frameworks, are prime examples of preexisting assets that can be imported from outside the organization and used to good advantage.

The “Make/Buy/Mine/Commission Analysis,” “Mining Existing Assets,” and “Using Externally Available Software” practice areas are most closely related to exploiting preexisting assets.

Three things, which are the outputs of the core asset development activity, are required for a production capability to develop products:

1. product line scope
2. core asset base
3. production plan

These items are described below.

1. **product line scope:** The product line scope is a description of the products that will constitute the product line or that the product line is capable of including. In its simplest form, the scope may consist of an enumerated list of product names. More typically, this description is cast in terms of the products’ similarities and differences. These similarities and differences might include the features or operations they provide, the performance or other quality attributes they exhibit, the platforms on which they run, and so forth.

   For a product line to be successful, its scope must be defined carefully. If the scope is too large and product members vary too widely, the core assets will be strained beyond their ability to accommodate the variation, economies of production will be lost, and the product line will collapse into an old-style, one-at-a-time product development effort. If the scope is too small, the core assets might not be built in a generic enough fashion to accommodate future growth, and the product line will stagnate: economies of scope will never be realized, and the full potential return on investment (ROI) will never materialize.

   The scope of the product line must target the right products, as determined by knowledge of similar products or systems, the prevailing or predicted market factors, the nature of competing efforts, and the organization’s business goals for embarking on a product line approach (such as market agility or merging a set of similar but currently independent product development projects).

   The scope of the product line evolves as market conditions change, as the organization’s plans change, as new opportunities arise, or as the organization, quite simply, becomes more adept at software product lines. Evolving the scope is the starting point for evolving the product line to keep it current.

   The scope definition of a product line is itself a core asset, evolved and maintained over the product line’s lifetime. Because it determines so much about the other core assets-in particular, what products they can support-we call it out separately. Scope is covered in more detail in the “Scoping” practice area.

2. **core asset base:** The core asset base includes all the core assets, which are the basis for the production of products in the product line. Not every core asset will necessarily be used in every product in the product line. However, all of them will be used in enough of the products to make their coordinated development, maintenance, and evolution pay off.

As we already described, these core assets almost certainly include an architecture that the products in the product line will share, as well as software components that are developed for systematic reuse across the product line. Any real-time performance models or other architecture evaluation results associated with the product line architecture are core assets. Software components may also bring with them test plans, test cases, and all manner of design documentation. Requirements specifications and domain models are core assets, as is the statement of the product line's scope. Web services and commercial off-the-shelf (COTS) software, if adopted, also constitute core assets. So do management artifacts such as schedules, budgets, and plans. Also, any production infrastructure such as domain-specific languages, tools, generators, and environments are core assets as well.

Each core asset should have an associated attached process that specifies how it will be used in the development of actual products. For example, the attached process for the product line requirements would give the process for expressing the requirements for an individual product. This process might simply say: (1) use the product line requirements as the baseline requirements, (2) specify the variation requirement for any allowed variation point, (3) add any requirements outside the set of specified product line requirements, and (4) validate that the variations and extensions can be supported by the architecture. The process might also specify the automated tool support for accomplishing these steps. Product constraints, production constraints, and the production strategy (contextual factors described above) influence the definition of attached processes. Those processes all follow an overall implementation approach, called the production method. The attached processes get folded into what becomes the production plan for the product line. Figure 3 illustrates this concept of attached processes and how they are incorporated into the production plan.

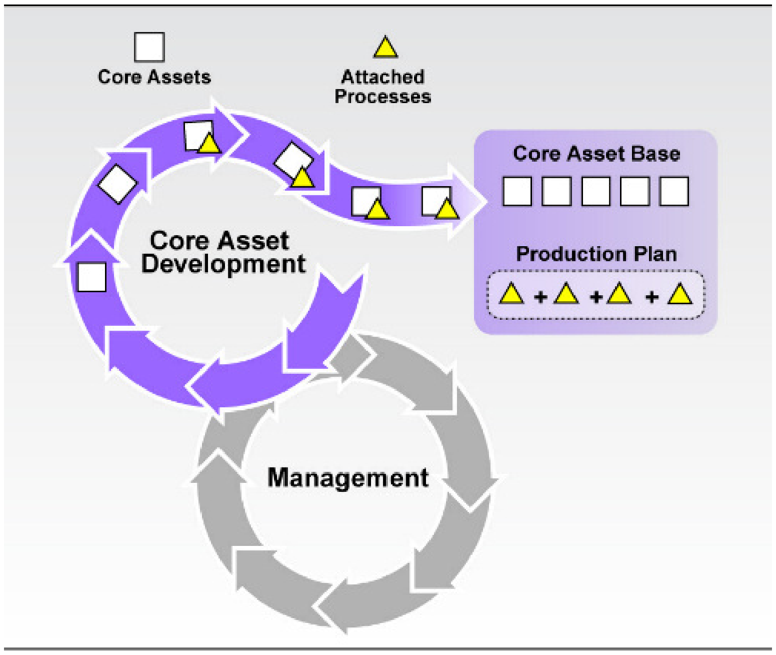


*Figure 3: Attached* Processes

There are also core assets of a less technical nature, such as the training specific to the product line, the business case for using a product line approach for this particular set of products, the technical management process definitions associated with the product line, and the set of identified risks for building products in the product line.

Finally, part of creating the core asset base is creating a concept of operations (CONOPS) that describes how the organization operates as a product line organization. The CONOPS defines (among other things) how that core asset base will be updated as the product line evolves, as more resources become available, as fielded products are maintained, and as technology changes or market shifts affect the product line scope.

3. **production plan:** A production plan prescribes how the products are produced from the core assets and fills two roles:
   - It includes the process to be used for building products (the *production process*). As noted above, each core asset should have an attached process that defines how it will be used in product development. The set of these attached processes, together with the necessary process "glue" to join them together into a coherent whole, define the production process for products. The attached processes and the (usually nontrivial) glue are designed to satisfy the production strategy and production constraints and reflect the chosen production method. The production method, which is the overall implementation approach, specifies the models, processes, and tools to be used in the attached processes across core assets. For example, variation could be achieved by selecting from an assortment of components or services to provide a given feature, by adding or deleting components, by tailoring one or more components via inheritance or parameterization, or by using aspects. It could also be the case that products are generated automatically, or at least in part. All of these are examples of production methods. The exact vehicle to be used to provide the requisite variation among products (by exercising the variation made available in the core assets) is described in the production process. Selecting incompatible variation mechanisms can undermine the efficiency of the production process. The "Technology Forecasting" and "Tool Support" practice areas influence the determination of the production method.
   - It lays out the project details to enable execution and management of the process and therefore includes such details as the schedule, bill of materials, and metrics. In practice, these details can be in a separate document but, conceptually, are a part of the production plan.

Figure 4 refines the intuitive notion of the production plan illustrated in Figure 3. It identifies the combined set of attached processes as the production process, calls out the project details, and explicitly shows the influence of the contextual factors (product constraints, production strategy, and production constraints) and the production method on the production plan. The "Process Discipline" and "Technical Planning" practice areas provide more information about the production plan and how to create it. It should be obvious that the production plan is, itself, an important core asset.

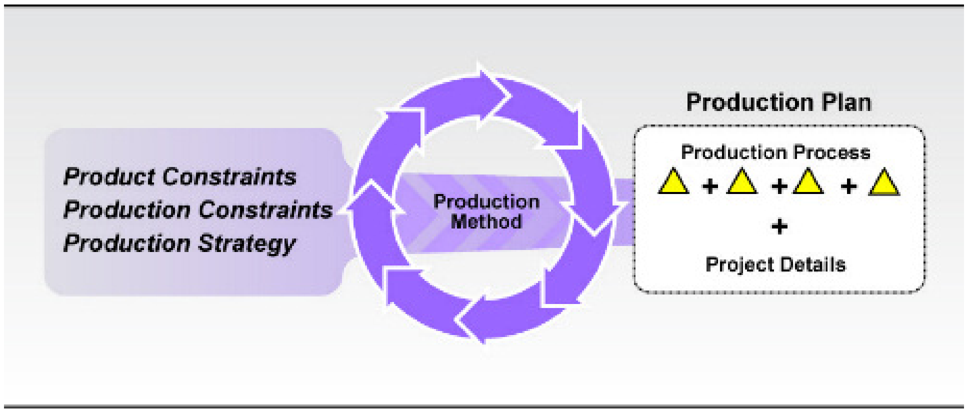


*Figure 4: Production Plan*

As described in Product Development, these three outputs (the product line scope, core asset base, and production plan) are necessary ingredients for feeding the product development activity, which turns out products that serve a particular customer or market niche.

## Product Development

The product development activity depends on the three outputs described above—the product line scope, the core assets, and the production plan—plus the product description for each individual product. Figure 5 illustrates these relationships.

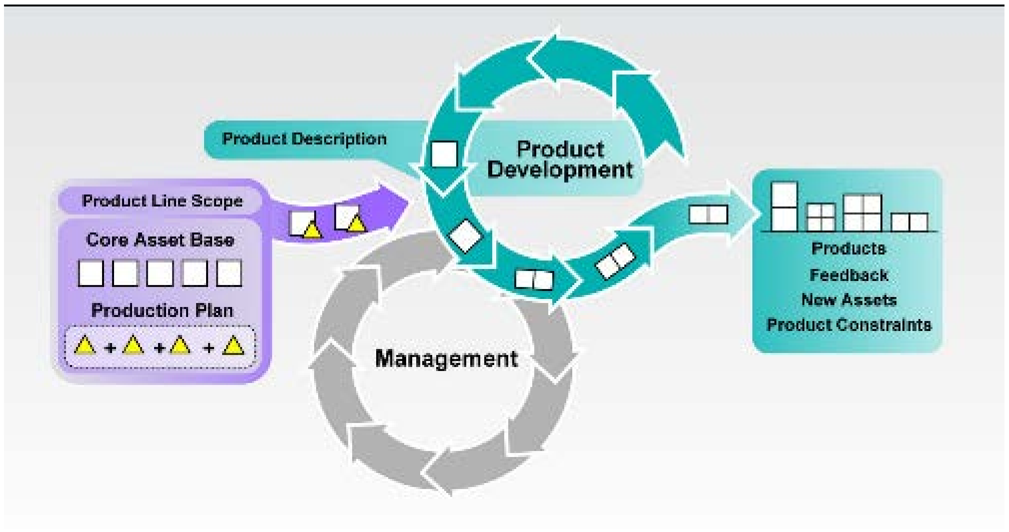


*Figure 5: Product Development*

Once more, the rotating arrows indicate iteration and intricate relationships. For example, the existence and availability of a particular product may well affect the requirements for a subsequent product. As another example, building a product that has previously unrecognized commonality with another product already in the product line will create pressure to update the core assets and provide a basis for exploiting that commonality for future products.

The inputs for the product development activity are as follows:

- the product description for a particular product, often expressed as a delta or variation from some generic product description contained in the product line scope (such a generic description is, itself, a core asset)
- the product line scope, which indicates whether it's feasible to include the product under consideration in the product line
- the core assets from which the product is built
- the production plan, which details how the core assets are to be used to build the product

Product builders use the core assets, in accordance with the production plan, to produce products that meet their respective requirements. Product builders also have an obligation to give feedback on any problems or deficiencies encountered with the core assets, so the core asset base remains healthy and viable.

## Management

Management plays a critical role in the successful fielding of a product line. Activities must be given resources and then be coordinated and supervised. Management at both the technical (or project) and organizational levels must be strongly committed to the software product line effort. That commitment manifests itself in a number of ways that feed the product line effort and keep it healthy and vital.

Organizational management identifies production constraints and ultimately determines the production strategy. Organizational management must create an organizational structure that makes sense for the enterprise and make sure that the organizational units receive the right resources (for example, well-trained personnel) in sufficient amounts. We define organizational management as the authority responsible for the ultimate success or failure of the product line effort. Organizational management determines a funding model that will ensure the evolution of the core assets and then provides the funds accordingly. Organizational management also orchestrates the technical activities in and iterations between the essential activities of core asset development and product development. Management should ensure that these operations and the communication paths of the product line effort are documented in an operational concept. At the organizational level, management mitigates those risks that threaten the success of the product line. The organization's external interfaces also need careful management. Product lines tend to engender different relationships with an organization's customers and suppliers, and these new relationships must be introduced, nurtured, and strengthened. One of the most important things that management must do is create an adoption plan that describes the desired state of the organization (that is, routinely producing products in the product line) and a strategy for achieving that state.

Technical management oversees the core asset development and product development activities by ensuring that those who build core assets and products are engaged in the required activities, follow the processes defined for the product line, and collect data sufficient to track progress. Technical management decides on the production method and provides the project management elements of the production plan.

Technical and organizational management also contribute to the core asset base by making available for reuse those management artifacts (especially schedules and budgets) used in developing products in the product line.

Finally, some individual or group should be designated to either fill the product line management role and act as a product line champion or find and empower one. That champion must be a strong, visionary leader who can keep the organization squarely pointed toward the product line goals, especially when the going gets rough in the early stages. Leadership is required for software product line success. Management and leadership are not always synonymous.

## All Three Together

Each of the three activities (core asset development, product development, and management) is individually essential, and their careful blending is also essential-a blend of technology and business practices. Different organizations may take different paths through the three activities. The path they take is a manifestation of their production strategy, as described in Core Asset Development. Product line practice patterns [Clements 2002c], especially the SEI *Adoption Factory* pattern [Northrop 2004a], are a way to help organizations chart their own path through the activities.

Many organizations begin a software product line by developing the core assets first. These organizations take a *proactive* approach [Krueger 2001a]. They define their product line scope to define the set (more often, a *space*) of systems that will constitute their product line. This scope definition provides a kind of mission statement for designing the product line architecture, components, and other core assets with the right built-in variation points to cover the scope. Producing any product within that scope becomes a matter of exercising the variation points of the components and architecture—that is, configuring the components and architecture—and then assembling and testing the system. Other organizations begin with one or a small number of products they already have and then use them to generate the product line core assets and future products. This approach is *reactive*.

Both of these approaches may be attacked incrementally. For example, a proactive approach may begin with the production of only the most important core assets, rather than all of them. Early products use those core assets. Subsequent products are built using more core assets as they are added to the collection. Eventually, the full core asset base is fielded; earlier products may or may not be reengineered to use the full collection. An incremental reactive approach works similarly; the core asset base is populated sparsely at first, using existing products as the source. More core assets are added as time and resources permit.

The proactive approach has obvious advantages—products come to market extremely quickly with a minimum of code writing. But there are also disadvantages. It requires a significant up-front investment to produce the architecture and the components that are generic (and reliable) across the entire product space. And it also requires copious up-front predictive knowledge—something that is not always available. In organizations that have long been developing products in a particular application domain, that is not a tremendous disadvantage. For a *green field* effort, where there is no experience or existing products, that is an enormous risk.

The reactive approach has the advantage of a much lower cost of entry to software product lines because the core asset base is not built up front. However, for the product line to be successful, the architecture and other core assets must be robust, extensible, and appropriate to future product line needs. If the core assets are not built beyond the ability to satisfy the specific set of products already in the works, extending them for future products may prove too costly.

Whatever the approach, the process is rarely, if ever, linear. The creation of products may have a strong feedback effect on the product line scope, the core assets, the production plan, and even the requirements for specific products. The ability to turn out a particular member of the product line quickly—perhaps a member that was not originally envisioned by the people responsible for defining the scope—will, in turn, affect the product line scope definition. Each new product may have similarities with other products that can be exploited by creating new core assets. As more products enter the field, efficiencies of production may dictate new system-generation procedures, causing the production plan to be reworked.

# Product Line Practice Areas

To achieve a software product line, you must carry out the three essential activities described in Product Line Essential Activities: core asset development, product development, and management. To be able to carry out the essential activities, you must master the practice areas relevant to each and apply them in a coordinated, focused fashion. By "mastering," we mean an ability to achieve repeatable, not just one-time, success.

A *practice area* is a body of work or a collection of activities that an organization must master to successfully carry out the essential work of a product line. Practice areas help to make the essential activities more achievable by defining activities that are smaller and more tractable than a broad imperative such as "develop core assets." Practice areas provide starting points from which organizations can make (and measure) progress in adopting a product line approach for software.

This framework defines the practice areas for product line practice. They all describe activities that are essential for any successful software development effort, not just software product lines. However, they all either take on particular significance or must be carried out in a unique way in a product line context. Those aspects that are specifically relevant to software product lines, as opposed to single-system development, are emphasized.

## Describing the Practice Areas

For each practice area, we present the following information:

- an introductory overview of the practice area that summarizes what it's about. You will not find a definitive discourse on the practice area here, since, in most cases, there is overlap with what can be found in traditional software engineering and management reference books. We provide a few basic references if you need a refresher.

- those aspects of the practice area that apply especially to a product line, as opposed to a single system. Here, you will learn in what ways traditional software and management practice areas need to be refocused or tailored to support a product line approach.
- how the practice area is applied to core asset development and product development, respectively. We separate these two essential activities in the framework. Although a given practice area typically applies to both of these broad areas, the focus will differ depending on whether you're building products or developing core assets.
- a description of any *example practices* that are known to apply to the practice area. An example practice describes a particular way of accomplishing the work associated with a practice area. Example practices are not meant to be end-to-end methodological solutions to carrying out a practice area but rather approaches to the problem that have been used in practice to build product lines. Whether a given example practice will work for your organization depends on the particular context of your situation.
- known risks associated with the practice area. These are ways in which a practice area can go wrong, to the detriment of the overall product line effort. Our understanding of these risks is borne out of the pitfalls of others in their product line efforts.
- where possible, a list of references for further reading, to support your investigation in areas where you desire more depth

## Organizing the Practice Areas

Since there are so many practice areas, we need a way of organizing them for easier access and reference. For this reason, we divide them loosely into three categories:

1. Software engineering practice areas are those necessary for applying the appropriate technology to create and evolve both core assets and products.
2. Technical management practice areas are those necessary for managing the creation and evolution of the core assets and the products.
3. Organizational management practice areas are those necessary for orchestrating the entire software product line effort.

Each of these categories appeals to a different body of knowledge and requires a different skill set for the people who must carry them out. The categories represent disciplines rather than job titles.

Figure 6 shows how the three categories are related.

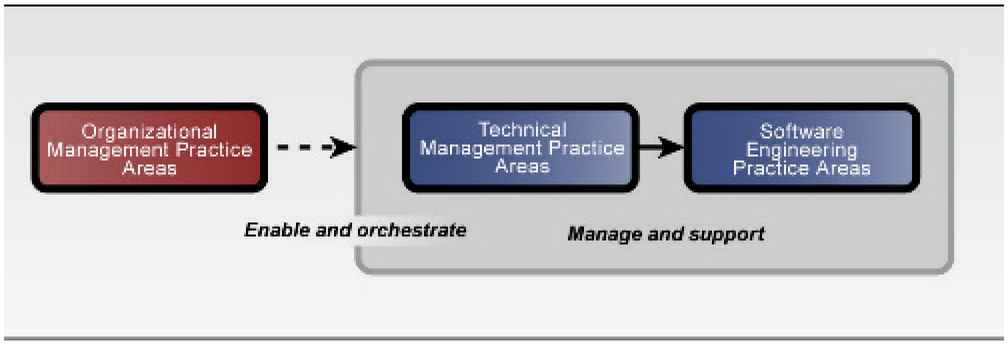

Organizational Management Practice Areas
Technical Management Practice Areas
Software Engineering Practice Areas
Enable and orchestrate
Manage and support

*Figure 6: Relationships Among Categories of Practice Areas*

There is no way to divide cleanly into practice areas the knowledge necessary to achieve a software product line. Some overlap is inevitable. We chose what we hope is a reasonable scheme and identified practice area overlap where possible.

The description of practice areas that follows is an encyclopedia; neither the ordering nor the categorization constitutes a method or an order for application. In other work, we provide product line practice patterns that show how to put the practice areas into play for a particular organization's context and goals [Clements 2002c, Northrop 2004a].

## Software Engineering Practice Areas

Software engineering practice areas are those necessary for applying the appropriate technology to create and evolve both core assets and products. They are

- architecture definition
- architecture evaluation
- component development
- mining existing assets
- requirements engineering
- software system integration
- testing
- understanding relevant domains
- using externally available software

### Architecture Definition

This practice area describes the activities that must be performed to define a software architecture. By software architecture, we mean the following:

*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. "Externally visible" properties are those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on [Bass 2003a].*

By making "externally visible properties" of *elements*[4] part of the definition, we intentionally and explicitly include elements' interfaces and behaviors, as part of the architecture. We will return to this point later. By contrast, design decisions or implementation choices that do not have system-wide ramifications or visibility are not architectural.

Architecture is key to the success of any software project. It is the first design artifact that begins to place requirements into a solution space. The quality attributes of a system (such as performance, modifiability, and availability) are, in large part, permitted or precluded by its architecture—if the architecture is not suitable from the beginning for these qualities, don't expect to achieve them by some miracle later. The architecture determines the structure and management of the development project as well as the resulting system, since teams are formed and resources allocated around architectural components. For anyone seeking to learn how the system works, the architecture is the place where understanding begins. The right architecture is absolutely essential for smooth sailing. The wrong one is a recipe for disaster.

**Architectural requirements:** For an architecture to be successful, its constraints must be known and articulated. And contrary to standard software engineering waterfall models, an architecture's constraints go far beyond implementing the required *behavior* of the system that is specified in a requirements document [Clements 2002c, p. 57]. Other architectural drivers that a seasoned architect knows to take into account include

- the quality attributes (as mentioned above) that are required for each product to be built from the architecture
- whether the system will have to interact with other systems
- the business goals that the developing organization has for the system. These might include ambitions to use the architecture as the basis for other systems (or even other software product lines). Or perhaps the organization wishes to develop a particular competence in an area such as Web-based database access. Consequently, the architecture will be strongly influenced by that desire.
- the best sources for components. A software architecture will call for a set of components to be defined, implemented, and integrated. Those components may be implemented in-house (see the "Component Development" practice area), be purchased or licensed from the commercial marketplace (see the "Using Externally Available Software" practice area), be contracted to third-party developers (see the "Developing an Acquisition Strategy" practice area), or be excavated

4 An *element* is a unit of software that has identity either at implementation time or at runtime. We use the term *component* to refer to a unit of software that must be developed or acquired as a unit.

from the organization's own legacy vaults (see the "Mining Existing Assets" practice area). The availability of preexisting components (ones that are commercial, open source, third party, or legacy) may influence the architecture considerably and cause the architect to carve out a place in the architecture where a preexisting component can fit, if doing so will save time or money or play into the organization's long-term strategies.

**Component interfaces:** As we said earlier in this section, architecture includes the interfaces of its components. It is therefore incumbent on the architect to specify those interfaces (or, if the component is developed externally, ensure that its interface is specified adequately by others). By *interface* we mean something far more complete than the simple functional signatures one finds in header files. Signatures simply name the programs and specify the numbers and types of their parameters, but they tell nothing about the semantics of the operations, the resources consumed, the exceptions raised, or the externally visible behavior. As Parnas wrote in 1972, an interface consists of the set of assumptions that users of the component may safely make about it-nothing more and nothing less [Parnas 1972a]. Approaches for specifying component interfaces are discussed in "Example Practices."

**Connecting components:** Applications are constructed by connecting components to enable communication and coordination. In simple systems that run on a single processor, the venerable procedure call is the oldest and most widely used mechanism for component interaction. In modern distributed systems, however, something more sophisticated is desirable. There are several competing technologies for providing these connections as well as other infrastructure services. Among the services provided by the infrastructures are

- remote procedure calls (allowing components to be deployed on different processors transparently)
- communication protocols
- object persistence
- the creation of standard methods, such as "naming services" or a "service discovery" that allow one component to find another via the component's registered name and/or services it provides.

These infrastructures, which are purchased as commercial packages, are components themselves that facilitate connection among other components. These infrastructure packages, like patterns, represent another class of already solved problems (highly functional component interactions for distributed systems) that the architect need not reinvent. Market contenders are Sun Microsystems' Java 2 Enterprise Edition (J2EE), including Enterprise Java Beans (EJB) (http://java.sun.com/j2ee), and Microsoft's .NET (http://www.microsoft.com/net).

**Architecture documentation and views:** In order for an architecture to achieve its effectiveness, it must be documented. Here, architectural views come into play. A view is a representation of a set of system elements and the relationships among them [Clements 2002a]. A view can be thought of as a projection of the architecture that includes certain kinds of information and suppresses other kinds; for example

- A *module decomposition* view shows how the software for the system is hierarchically decomposed into smaller units of implementation.

- A *communicating-processes* view shows the processes in the software and how they communicate or synchronize with each other but does not show how the software is divided into layers (if it is).
- A *layered view* shows how the software is divided into layers but does not show the processes involved.
- A *deployment* view shows how software is assigned to hardware elements in the system.

There are many views of an architecture; choosing which ones to document is a matter of what information you wish to convey. Each view has a particular usefulness to one or more segments of the stakeholder community [IEEE 2000a] and should be chosen and engineered with that in mind.

## Aspects Peculiar to Product Lines

All architectures are abstractions that admit a plurality of instances; a great source of their conceptual value is, after all, the fact that they allow us to concentrate on design while admitting a number of implementations. But a product line architecture goes beyond this simple dichotomy between design and code—it is concerned with identifying and providing mechanisms to achieve a set of explicitly allowed variations (because when exercised, these variations become products). Choosing appropriate variation mechanisms may be among the product line architect's most important tasks. The variation mechanisms chosen must support

- the variations reflected in the products. The product constraints (see Core Asset Development) and the result of a scoping exercise (see the "Scoping" practice area) provides information about envisioned variations in the products of the product line—variations that will need to be supported by the architecture. These variations often manifest as different quality attributes. For example, a product line may include both a high-performance product with enhanced security features and a low-end version of the same product.
- the production strategy and production constraints (as described in Core Asset Development). The variation mechanisms provided by the architecture should be chosen carefully, so they support the way the organization plans to build products.
- efficient integration. Integration may assume a greater role for software product lines than for one-off systems simply because of the number of times it's performed. A product line with a large number of products and upgrades requires a smooth and easy process for each product. Therefore, it pays to select variation mechanisms that allow for reliable and efficient integration when new products are turned out. This need for reliability and efficiency means some degree of automation. For example, if the variation mechanism chosen for the architecture is component selection and deselection, you will want an integration tool that carries out your wishes by selecting the right components and feeding them to the compiler or code generator. If the variation mechanism is parameterization or conditional compilation, you will want an integration tool that checks the parameter values for consistency and compatibility before it feeds those values to the compilation step. Hence, the variation mechanism chosen for the architecture will go hand in hand with the integration approach (see the "Software System Integration" practice area).

Support for variation can take many forms (and be exercised many times [Clements 2002c, p. 64]). Mechanisms to achieve variation in the architecture are discussed under “Example Practices.”

Products in a software product line exist simultaneously and may vary from each other in terms of their behavior, quality attributes, platform, network, physical configuration, middleware, and scale factors and in a multitude of other ways. Each product may well have its own architecture, which is an instance of the product line architecture achieved by exercising the variation mechanisms. Hence, unlike an organization engaged in single-system development, a product line organization will have to manage many related architectures simultaneously.

There must be documentation for the product line architecture as it resides in the core asset base and for each product’s architecture (to the extent that it varies from the product line architecture). For the product line architecture, the views need to show the variations that are possible and must describe the variation mechanisms chosen with the rationale for the variation. Furthermore, a description—the attached process—is required that explains how to exercise the mechanisms to create a specific product. The views of the product architectures, on the other hand, have to show how those variation mechanisms have been used to create this product’s architecture. As with all core assets, the attached process becomes the part of the production plan that deals with the architecture.

### Application to Core Asset Development

The product line architecture is an early and prominent member in the collection of core assets. The architecture is expected to persist over the life of the product line and to change relatively little and slowly over time. The architecture defines the set of software components (and hence their supporting assets such as documentation and test artifacts) that populates the core asset base. The product line architecture—together with the production plan—provides the prescription (harkening to the “in a prescribed way” from the definition of a software product line) for how products are built from core assets.

### Application to Product Development

Once it’s been placed in the core asset base for the product line, the architecture is used to create product architectures for each new product according to the architecture’s attached process. If the product builders discover a variation point or a needed mode of variation that is not permitted by the architecture, they should bring it to the architect’s attention; if the variation is within the product line’s scope (or deemed desirable to add to the scope), the architecture may be enhanced to accommodate it. The “Operations” practice area deals with setting up this feedback loop in the organization.

### Example Practices

We categorize example practices for architecture definition into those concerned with understanding the requirements for the architecture; designing the architecture; and communicating or documenting the architecture.

## Understanding the Requirements for the Architecture

**Quality Attribute Workshops:** Prerequisite to designing an architecture is understanding the behavioral and quality attribute requirements that it must satisfy. One way to elicit these requirements from the architecture's stakeholders is with an SEI Quality Attribute Workshop (QAW) [SEI 2007g]. QAWs provide a method for identifying the quality attributes that are critical to a system architecture—attributes such as availability, performance, security, interoperability, and modifiability. In the QAW, an external team facilitates meetings between stakeholders during which scenarios representing the quality attribute requirements are generated, prioritized, and refined (i.e., adding additional details such as the participants and assets involved, the sequence of activities, and questions about quality attribute requirements). The refined scenarios can be used in different ways; for example, as seed scenarios for an evaluation exercise or as test cases in an acquisition effort.

**Use of the production strategy:** The choice of variation mechanisms is strongly influenced by the organization's production strategy. That strategy describes how the organization plans to build the specific products from the core assets. For example, an organization may decide that an integration team will assemble a product by selecting from existing components. This strategy forces the architecture team to focus on component substitution as a variation mechanism. Mechanisms that require additional coding may not be appropriate in this setting.

**Planning for architectural variation:** Nokia has used a "requirements definition hierarchy" as a way to understand what variations are important to particular products [Kuusela 2000a]. The hierarchy consists of design objectives (goals or wishes) and design decisions (solutions adopted to meet the corresponding goals). For example, a design objective might be "the system shall be highly reliable." One way to meet that objective is to decree that "the system shall be a duplicated system." That, in turn, might mean that "the system shall have duplicated hardware" and/or "the system shall duplicate communication links." Another way to meet the reliability objective is to decree that "the system shall have a self-diagnostic capacity," which can be met in several ways. Each box in the hierarchy is tagged with a vector, each element of which corresponds to a product in the product line. The value of an element is the priority or importance given to that objective, or to the endorsement of that design decision, by the particular product. For example, if an overall goal for a product line is high reliability, being a duplicated system might be very important to Product 2 and Product 3 but not at all important to Product 1 (a single-chip system).

The requirements definition hierarchy is a tool that the architect can use as a bridge between the product line's scope (see the "Scoping" practice area), which will tell what variations the architecture will have to support, and the architecture, which may support the variation in a number of ways. It is also useful to see how widely used a new feature or variation will be: should it be incorporated into the architecture for many products to use, or is it a one-of-a-kind requirement best left to the devices of the product that spawned it? The hierarchy is a way for the architect to capture the rationale behind such decisions.

## Designing the Architecture

**Architecture definition and architecture-based development:** As the field of software architecture has grown and matured, methods of creating, defining, and using architecture have proliferated. Many example practices related to architecture definition are defined in widely available works [Kruchten 1998a, Jacobson 1997a, Hofmeister 2000a, Bachmann 2000a]. The Rational Unified Process (RUP) is a method used for object-oriented systems. A good resource for RUP is the book The Rational Unified Process: An Introduction [Kruchten 2004a].

**Attribute-Driven Design (ADD):** The SEI Attribute-Driven Design (ADD) method [SEI 2007a] is a method for designing the software architecture of a product line to ensure that the resulting products have the desired qualities. ADD is a recursive decomposition method that starts by gathering architectural drivers that are a combination of the quality, functional, and business requirements that "shape" the architecture. The steps at each stage of the decomposition are

1. **Choose architectural drivers:** The architectural drivers are the combination of quality, business, and functional goals that "shape" the architecture.
2. **Choose patterns and children component types to satisfy drivers:** There are known patterns to achieve various qualities. Choose the solutions that are most appropriate for the high-priority qualities.
3. **Instantiate children design elements and allocate functionality from use cases using multiple views:** The functionality to be achieved by the product family is allocated to the component types.
4. **Identify commonalities across component instances:** These commonalities are what define the product line, as opposed to individual products.
5. **Validate that the quality and functional requirements and any constraints have not been precluded from being satisfied by the decomposition.**
6. **Refine use cases and quality scenarios as constraints to children design elements:** Because ADD is a decomposition method, the inputs for the next stage of decomposition must be prepared.

**Architectural patterns:** Architectures are seldom built from scratch but rather evolve from solutions previously applied to similar problems. Architectural patterns represent a current approach to reusing architectural design solutions. An architecture pattern[5] is a description of component types and a pattern of their runtime control and/or data transfer [Shaw 1996a]. Architectural patterns are becoming a de facto design language for software architectures. People speak of pipe-and-filter, n-tier, client-server, or agent-based architectures, and these phrases immediately convey complex and sophisticated design information. Architectural pattern catalogues exist that explain the properties of a particular pattern, including how well-suited each one is for achieving specific quality attributes such as security or high performance. Buschmann, Schmidt, and colleagues provide examples of catalogs [Buschmann

5 The term used by Shaw and Garlan was architectural style, which is synonymous with *pattern* [Shaw 1996a].

1996a, Schmidt 2000a]. Using a previously catalogued pattern shortens the architecture definition process, because patterns come with pedigrees: what applications they work well for, what their performance properties are, where they can easily accommodate variation points, and so forth. Product line architects should be familiar with well-known architectural patterns as well as patterns (well-known or not) that are used in systems similar to the ones they are building.

**Service-oriented architectures:** One architectural pattern that's very popular now is the service-oriented architecture. A service-oriented architecture is one in which the components are services. A service is a reusable, self-contained, distributed component with a published interface that stresses interoperability, is usually thread-safe, and is discoverable and dynamically bound. Service-oriented systems work by "stringing together" services that have specific, well-defined functionality into chains that do sophisticated, useful work. Services can be locally developed or (in theory) "discovered" on a company's intranet or even on the World Wide Web and bound at runtime. Standards exist for services communicating with each other via messaging based on the Extensible Markup Language (XML), for specifying what services do, and for quality-of-service contracts necessary to insure that a service provides the level of functionality and quality attributes required. A service-oriented architecture's basic variation mechanism is component replacement—that is, choosing different services or stringing together services in a different way to meet the needs of individual products.

**Aspect-oriented software development (AOSD):** AOSD is an approach to program development that makes it possible to modularize systemic properties of a program such as synchronization, error handling, security, persistence, resource sharing, distribution, memory management, replication, and the like that would otherwise be distributed widely across the system, making it hard to change. An *aspect* is a special kind of module that implements one of these specific properties that would otherwise cut across other modules. As that property varies, the effects "ripple" through the entire program automatically. As an example, an AOSD program might define "the public methods of a given package" as a crosscutting structure and then say that all those methods should do a certain kind of error handling. This aspect would be coded in a few lines of well-modularized code. AOSD is an architectural approach, because it provides a means of separating concerns that would otherwise affect a multitude of components constructed to separate a different, orthogonal set of concerns. AOSD is appealing for product lines, because the variations can often be represented as aspects. A good starting point for understanding AOSD is provided at http://aosd.net.

**Mechanisms for achieving variability in a product line architecture (1):** Svahnberg and Bosch created a list of variability mechanisms for product lines that includes mechanisms for building variability into components. Svahnberg and Bosch also include these architectural mechanisms [Svahnberg 2000a]:

- **configuration and module interconnection languages:** used to define the build-time structure of a system, including selecting (or deselecting) whole components
- **generation:** used when there is a higher level language that can be used to define a component's desired properties
- **compile-time selection of different implementations:** The variable *#ifdefs* can be used when variability in a component can be realized by choosing different implementations.

Code-based mechanisms used to achieve variability within individual components are discussed in the "Component Development" practice area.

**Mechanisms for achieving variability in a product line architecture (2):** Philips Research Laboratories uses *service component frameworks* to achieve diversity in its product line of medical imaging systems [Wijnstra 2000a]. Goals for that family include extensibility over time and support for different functions at the same time. A framework is a skeleton of an application that can be customized to yield a product. White-box frameworks rely heavily on inheritance and dynamic binding; knowledge of the framework's internals is necessary in order to use it. Black-box frameworks define interfaces for components that can be plugged in via composition tools. A service component framework is a type of black-box framework that supports a variable number of plug-in components. Each plug-in is a container for one or more services, which provide the necessary functionality. All services support the framework's defined interface but exhibit different behaviors. Clients use the functionality provided by the component framework and the services as a whole; the assemblage is, itself, a component in the products' architecture. Conversely, units in the product line architecture may consist of or contain one or more component frameworks.

**Mechanisms for achieving variability in a product line architecture (3):** Bachmann and Clements sum up the current approaches for variation mechanisms in product line architectures and sketch an economics-based approach for choosing them [Bachmann 2005a].

### Communicating and Documenting the Architecture

**Architecture documentation:** Recently, in the software engineering community, more attention has been paid to writing down a software architecture so that others can understand it, use it to build systems, and sustain it. The Unified Modeling Language (UML) is the most-often-used formal notation for software architectures, although it lacks many architecture-centric concepts. The SEI developed the Views and Beyond approach to documentation [Clements 2002a], which holds that documenting a software architecture is a matter of choosing the relevant views based on projected stakeholder needs, documenting those views, and then documenting the information that applies across all of them. Examples of cross-view information include how the views relate to each other and stakeholder-centric roadmaps through the documentation that let people with different interests find information relevant to them quickly and efficiently. The approach includes a three-step method for choosing the best views to engineer and document for any architecture, and the overall approach produces a result compliant with the Institute of Electrical and Electronics Engineers' (IEEE's) recommended best practice on documenting architectures of software-intensive systems [IEEE 2000a].

**Specifying component interfaces:** Interfaces are often specified using a contractual approach. Contracts state pre- and postconditions for each service and define invariants that express constraints about the interactions of services within the component. The contract approach is static and does not address the dynamic aspects of a component-based system or even the dynamic aspects of a single component's behavior. Additional techniques such as state machines [Harel 1998a] and interval temporal logic [Moszkowski 1986a] can be used to specify constraints on the component that deal with the ordering of events and the timing between events. For example, a service may create a thread and assign it work to do that will not be completed within the service's execution window. A postcondition for that service would include the logical clause for "eventually this work is accomplished."

A complete contract should include information about what will be both provided and required. The typical component interface specification describes the services that a component provides. To fully document a component so that it can be integrated easily with other components, the specification should also document the resources that the component requires. In addition, this documentation provides a basis for determining whether there are possible conflicts between the resources needed for the set of components that make up the application.

A component's interface provides only a specification of how individual services respond when invoked. As components are integrated, additional information is needed. The interactions between two components needed to achieve a specific objective can be described as a *protocol*. A protocol groups together a set of messages from both components and specifies the order in which they are to occur.

Each component exhibits a number of externally visible attributes that are important to its use but are often omitted (incorrectly) from its interface specification. Performance (throughput) and reliability are two such attributes. The standard technique for documenting the performance of a component is the computational complexity of the dominant algorithms. Although this technique is platform independent, it is difficult to use in reasoning about satisfying requirements in real-time systems, because it fails to yield an actual time measure. Worse, it uses information that will change when algorithms (presumably encapsulated within the component) change. A better approach is to document performance bounds, setting an upper bound on the time consumed. The documentation remains true when the software is ported to a platform at least as fast as the current one—a safe assumption in today's environment. Cases in which the stated bounds are not fast enough can be resolved on a case-by-case basis. If the product can indeed meet the more stringent requirement on that product's platform, that fact can be revealed. If it cannot, either remedial action must be taken or the requirement must be relaxed.

## Practice Risks

The biggest risk associated with this practice area is failing to have a suitable product line architecture. An unsuitable product line architecture will result in

- components that do not fit together or interact properly
- products that do not meet their behavioral, performance, or other quality attribute goals
- products that should be within scope but which cannot be produced from the core assets at hand
- a tedious and ad hoc product-building process

These effects, in turn, will lead to extensive and time-consuming rework, poor system quality, and an inability to realize the product line's full benefits. If product teams do not find the architecture suitable for their products and easy to understand and use, they may bypass it, resulting in the eventual degradation of the entire product line concept.

Unsuitable architectures could result from

- **the lack of a skilled architect:** A product line architect must be skilled in current and promising technologies, the nuances of the application domains at hand, modern design techniques and tool support, and professional practices such as the use of architectural patterns. The architect must know all the sources of requirements and constraints on the architecture, including those not traditionally specified in a requirements specification (such as organizational goals) [Clements 2002c, p. 58].
- **the lack of sound input:** The product line scope and production strategy must be well-defined and stable. The requirements for products must be articulated clearly and completely enough for architectural decisions to be reliably based on them. Forthcoming technology, which the architecture must be poised to accept, must be forecast accurately. Relevant domains must be understood so that their architectural lessons are learned. To the extent to which the architect is compelled to make guesses, the architecture poses a risk.
- **poor communication:** The best architecture is useless if it is documented and communicated in ways that its consumers—for example, the product builders—cannot understand. An architecture whose documentation is chronically out of date is effectively the same as an undocumented architecture. Clear and open two-way communication channels must exist between the architect and the organizations using the architecture. Architecture documentation that is appropriate for architects and developers may not be good enough for other stakeholders, who, for example, may not understand UML diagrams.
- **a lack of supportive management and culture:** Management must support the creation and use of the product line architecture, especially if the architecture group is separate from the product development group. Failing this, product groups may “go renegade” and make unilateral changes to the architecture, or decline to use it at all, when turning out their systems. There are additional risks if management does not support the strong integration of system and software engineering.
- **architecture in a vacuum:** The exploration and definition of software architecture cannot take place in a vacuum separate from system architecture.
- **poor tools:** There are precious few tools for this practice area, especially those that help with designing, specifying, or exercising an architecture’s variation mechanisms—a fundamental part of a product line architecture. Tools for testing the compliance of products to an architecture are virtually nonexistent.
- **poor timing:** Declaring that an architecture is ready for production before it really is leads to stagnation, while declaring it too late may allow unwanted variation. Discretion is needed when deciding when and how firmly to freeze the architecture. The time required to fully develop the architecture also may be too long. If product development is curtailed while the product line architecture is being completed, developers may lose patience, management may lose resolve, and salespeople may lose market share.

Unsuitable architectures are characterized by

- **inappropriate parameterization:** Overparameterization can make a system unwieldy and difficult to understand. Underparameterization can eliminate some of the necessary system customizations. The early binding of parameters can also preclude easy customization, while the late binding of parameters can lead to inefficiencies.

- **inadequate specifications:** Components may not integrate properly if their specifications are sketchy or limited to static descriptions of individual services.
- **decomposition flaws:** Without an appropriate decomposition of the required system functionality, a component may not provide the functionality needed to implement the system correctly.
- **wrong level of specificity:** A component may not be reusable if the component is too specific or too general. If the component is made so general that it encompasses multiple domain concepts, the component may require complex configuration information to make it fit a specific situation and therefore be inherently difficult to reuse. The excessive generality may also tax performance and other quality attributes to an unacceptable point. If the component is too specific, there will be few situations in which it is the correct choice.
- **excessive intercomponent dependencies:** A component may become less reusable if it has excessive dependencies on other components.

## Further Reading

*General software architecture:*

[Bass 2003a]
Bass, Clements, and Kazman emphasize architecture's role in system development and provide several case studies of architectures used to solve real problems. One is an architecture for the CelsiusTech ShipSystem 2000 product line. Their book also includes an extensive discussion of architectural views.

[Buschmann 1996a] & [Schmidt 2000a]
Buschmann, Schmidt, and colleagues provide excellent examples of architectural patterns.

[Hofmeister 2000a]
Hofmeister emphasizes views and structures and provides a solid treatment of building a system from an architecture and its views.

[SEI 2007b]
The SEI's Software Architecture Technology (SAT) Web site provides a wide variety of software architecture resources and links.

[Shaw 1996a]
Shaw and Garlan provide an excellent treatment of architectural patterns (what they call styles) and their ramifications for system building.

*Product line architecture:*

[Bosch 2000a]
Bosch brings a dedicated product line focus to the mix. His work is required reading for the product line practitioner.

*Software architecture from a strictly object-oriented point of view:*

[Booch 1994a]
Booch offers a good foundation for architecture definition.

[Buschmann 1996a]
The work of Buschmann and colleagues raises the design pattern phenomenon to the arena of software architecture and is a good staple in any architect's toolbox.

[Jacobson 1997a]
Jacobson, Griss, and Jonsson devote an entire section to architectural patterns for object-oriented systems designed with strategic reuse in mind.

[Kruchten 2004a]
Kruchten's book is a good source for gaining an understanding of RUP.

[Smith 2001a]
Smith and Williams' book contains three chapters on principles and guidance for architecting systems (object-oriented or not) in which performance is a concern.

*Problem solving:*

[Jackson 2000a]
Jackson classifies, analyzes, and structures a set of recurring software development problems, organized according to how the software will interact with the outside world.

*Architecture documentation:*

[Clements 2002a]
Clements and colleagues explain the Views and Beyond approach to architecture documentation.

*UML:*

[OMG 2007a]
This Web site is the starting point for an investigation of UML.

## Architecture Evaluation

"Marry your architecture in haste, and you can repent in leisure." So admonished Barry Boehm in a 2000 lecture [Boehm 2000a]. The architecture of a system represents a coherent set of the earliest design decisions, which are the most difficult to change and the most critical to get right. The architecture is the first design artifact that addresses the quality goals of the system such as security, reliability, usability, modifiability, and real-time performance. The architecture describes the system structure and serves as a common communication vehicle among the system stakeholders: developers, managers, maintainers, users, customers, testers, marketers, and anyone else who has a vested interest in the development or use of the system.

With the advent of repeatable, cost-effective architecture evaluation methods, it is now feasible to make architecture evaluation a standard part of the development cycle. And because so much rides on the architecture and it is available early in the life cycle, it makes utmost sense to evaluate the architecture early when there is still time for mid-course correction. In any nontrivial project, there are competing requirements and corresponding architectural decisions that must be made to resolve them. It is best to air and evaluate those decisions and then document the basis for making them before the decisions are cast into code. Architecture evaluation is a form of artifact validation, just as software testing is a form of code validation. In the “Testing” practice area, we discuss the validation of artifacts in general—including design models such as the architecture—but the architecture for the product line is so foundational that we give its validation its own special practice area.

The evaluation can be done at a variety of stages during the design process; for example, when the architecture is still on the drawing board and candidate structures are being weighed. The evaluation can also be done later, after preliminary architectural decisions have been made but before detailed design has begun. The evaluation can even be done after the entire system has been built (such as in the case of a reengineering or mining operation). The outputs will depend on the stage at which the evaluation is performed. Enough design decisions must have been made so that the achievement of the requirements and quality attribute goals can be analyzed. The more architectural decisions that have been made, the more precise the evaluation can be. On the other hand, the more decisions that have been made, the more difficult it is to change any one of them.

An organization’s business goals for a system lead to particular behavioral requirements and quality attribute goals. The architecture is evaluated with respect to those requirements and goals. Therefore, before an evaluation can proceed, the behavioral and quality attribute goals against which an architecture is to be evaluated must be made explicit. These quality attribute goals support the business goals. For example, if a business goal is that the system should be long-lived, modifiability becomes an important quality attribute goal.

Quality attribute goals, by themselves, are not definitive enough for either design or evaluation; they must be made more concrete. Using modifiability as an example, if a product line can be adapted easily to have different user interfaces but is dependent on a particular operating system, is it modifiable? The answer is “yes” with respect to the user interface but “no” with respect to porting to a new operating system. Whether this architecture is suitably modifiable depends on what modifications to the product line are expected over its lifetime. That is, the abstract quality goal of modifiability must be made concrete: modifiable with respect to what kinds of changes, exactly? The same is true for other attributes. The evaluation method that you use must include a way to concretize the quality and behavioral goals for the architecture being evaluated.

## Aspects Peculiar to Product Lines

In a product line, architecture assumes a dual role. There is the architecture for the product line as a whole, and there are architectures for each of the products. The latter are produced from the former by exercising the built-in variation mechanisms according to the production plan. All the architectures should be evaluated. The product line architecture should be evaluated for its robustness and generality to make sure it can serve as the basis for products in the product line’s envisioned scope. Product architectures should be evaluated to make sure they meet the specific behavioral and quality requirements of the product at hand. In practice, the extent to which product architectures require separate evaluations depends on the extent to which they differ from the product line architecture and on the degree of automation used in creating them.

Often, some of the hardware and other performance-affecting factors for a product line architecture are unknown to begin with. Thus, the evaluation of the product line architecture must establish bounds on the performance that the architecture is able to achieve, assuming bounds on hardware and other variables. Evaluating the product line architecture identifies potential contention problems and helps put in place the policies and strategies to resolve them. The evaluation of a particular instance of the product line architecture can verify whether the hardware and performance decisions that have been made are compatible with the goals of that instance.

### Application to Core Asset Development

Clearly, an evaluation should be applied to the core asset that is the product line architecture.

Some of the business goals (against which the product line architecture will be evaluated) will be related to the fact that the architecture is for a product line. For example, the architecture will almost certainly have built-in variation points that can be exercised to derive specific products having different attributes. The evaluation will have to focus on the variation points to make sure they are appropriate, offer sufficient flexibility to cover the product line’s intended scope, can be exercised in a way that lets products be built quickly to support the product line’s production constraints (see Core Asset Development), and do not impose unacceptable runtime performance costs. Also, different products in the product line may have different quality attribute requirements, and the architecture will have to be evaluated for its ability to provide all the required combinations.

As the requirements, business goals, and architecture all evolve over time, periodic (although not frequent) reevaluations should be performed to discover whether the architecture and business goals are still well matched. Those reevaluations may be shortened by focusing on the important differences between the business goals of the product and those of the overall product line. Such reevaluations should reveal important architectural differences to key on. Some evaluation methods produce a report that summarizes what the articulated, prioritized, quality attribute goals are for the architecture and how the architecture satisfies them. Such a report makes an excellent rationale record, which can then accompany the architecture throughout its evolution as a core asset in its own right.

An architecture evaluation can also be performed on components that are candidates to be acquired as core assets, as well as on components developed in-house. In either case, the evaluation proceeds with the help of technical personnel from the organization that developed the software. An architecture evaluation is not possible for “black-box” architecture acquisitions where the architecture is not visible. The quality attribute goals to be used for the evaluation will include how well the potential acquisition will (1) support the quality goals for the product line and (2) evolve over time to support the intended evolution of the products in the product line.

Because product architectures are variations of the product line architecture, the product architecture evaluation is similarly a variation of the product line architecture evaluation. The artifacts produced during both product line architecture and product architecture evaluations (such as scenarios, checklists, and so on) will certainly have reuse potential and may become core assets by themselves.

## Application to Product Development

An architecture evaluation should be performed on an instance or variation of the architecture that will be used to build one or more of the products in the product line. The extent to which it is a separate, dedicated evaluation depends on the extent to which the product architecture differs in quality-attribute-affecting ways from the product line architecture or on how much the instantiation process can be trusted to produce a product architecture with the required quality attributes. If the differences are minor or exercising the variation mechanisms will most likely produce the expected results, these product architecture evaluations can be abbreviated. The results of architecture evaluation for product architectures often provide useful feedback to the architect(s) of the product line architecture and fuel improvements in it.

Finally, when a new product is proposed that falls outside the scope of the original product line (for which the architecture was presumably evaluated), the product line architecture can be reevaluated to see if it will suffice for this new product. If it will, the product line's scope is expanded to include the new product. If it will not, the evaluation can be used to determine how the architecture would have to be modified to accommodate the new product.

## Example Practices

Several different architecture evaluation techniques exist and can be modified to serve in a product line context. Techniques can be categorized broadly as either questioning techniques (those using questionnaires, checklists, scenarios, and the like as the basis for architectural investigation) or measuring techniques (such as simulation or experimentation with a running system) [Abowd 1996a]. Well-versed architects should have a spectrum of techniques in their evaluation kit. For full-fledged architectures, software performance engineering or a method such as the SEI Architecture Tradeoff Analysis Method (ATAM) or the SEI Software Architecture Analysis Method (SAAM) is indispensable. For less complete designs, a technique such as SEI Active Reviews for Intermediate Designs (ARID) is handy. For validating architectural (and other design) specifications, active design reviews (ADRs) are helpful. The article "The Bibliography of Software Architecture Analysis" published in *Software Engineering Notes* provides more alternatives [Zhao 1999a].

**ATAM:** The ATAM is a scenario-based architecture evaluation method that focuses on a system's quality goals. The input to the ATAM consists of an architecture, the business goals of a system or product line, and the perspectives of stakeholders involved with that system or product line. The ATAM focuses on an understanding of the architectural approach that is used to achieve particular quality goals and the implications of that approach. The method uses stakeholder perspectives to derive a collection of scenarios that give specific instances for usage, performance requirements, various types of failures, possible threats, and a set of likely modifications. The scenarios help the evaluators understand the inherent architectural risks, sensitivity points to particular quality attributes, and tradeoffs among quality attributes.

Of particular interest to ATAM-based evaluations of product line architectures are the sensitivity points to extensibility (or variation) and the tradeoffs of extensibility with other quality attribute goals (usually real-time performance, security, and reliability).

The output of an ATAM evaluation includes

- the collection of scenarios that represent the stakeholders' highest priority expression of usage and quality attribute goals for the system and its architecture
- a utility tree that assigns specific scenarios to the quality attributes that apply to the system architecture being evaluated
- specific analysis results, including the explicit identification of sensitivity points, tradeoffs, and other architectural decisions that impact the desired quality attributes either positively or negatively. Decisions that have a negative impact constitute areas of risk.

The ATAM can be used to evaluate both product line and product architectures at various stages of development (conceptual, before code, during development, or after deployment). An ATAM evaluation usually requires three full days plus some preparation and preliminary investigation time. The ATAM is described in detail by Clements, Kazman, and Klein [Clements 2001a] and on the SEI's Software Architecture Technology (SAT) Web site [SEI 2007b].

**SPE:** Software performance engineering (SPE) is a method for making sure that a design will allow a system to meet its performance goals before it has been built. SPE involves articulating the specific performance goals, building coarse-grained models to get early ideas about whether the design is problematic and refining those models along well-defined lines as more information becomes available. Conceptually, SPE resembles the ATAM but the singular quality attribute of interest is performance. Smith wrote the definitive resource for SPE [Smith 1990a] and, along with Woodside, its concise method description [Gelenbe 1999a].

**ARID:** ARID is a hybrid design review method that combines the active design review philosophy of ADRs with the scenario-based analysis of the ATAM and SAAM [Clements 2000a]. ARID was created to evaluate partial (for example, subsystem) designs in their early or conceptual phases, before they are fully documented. While such designs are architectural in nature, they are not complete architectures. ARID works by assembling stakeholders for the design, having them adopt a set of scenarios that express a set of meaningful ways in which they would want to use the design, and then having them write code or pseudocode that uses the design to carry out each scenario. This process wrings out any conceptual flaws early and familiarizes stakeholders with the design before it is fully documented. An ARID exercise takes one to two days.

**ADRs:** An ADR [Parnas 2001a] is a technique that can be used to evaluate an architecture still under construction. ADRs are particularly well-suited for evaluating the designs of single components or small groups of components before the entire architecture has been solidified. The principle behind ADRs is that stakeholders are engaged to review the documentation that describes the interface facilities provided by a component and then are asked to complete exercises that compel them to actually use that documentation. For example, each reviewer may be asked to write a short code segment that performs some useful task using the component's interface facilities, or each reviewer may be asked to verify that essential information about each interface operation is present and well specified. ADRs are contrasted with unstructured reviews in which people are asked to read a document, attend a long meeting, and comment on whatever they wish. In an ADR, there is no meeting; reviewers are debriefed (or walked through their assignments) individually or in small informal groups. The key is to avoid asking questions to which a reviewer can blithely and without much thought answer “yes” or “no.” An ADR for a medium-sized component usually takes a full day from each of six or so reviewers, who can work in parallel. The debriefing takes about an hour for each session.

## Practice Risks

The major risk associated with this practice is failing to perform an effective architecture evaluation; such an evaluation prevents unsuitable architectures from being allowed to pollute a software product line effort. Architecture evaluation is the safety valve for product line architectures, and an ineffective evaluation will lead to the same consequences as an unsuitable architecture. (For a list of these consequences, see the “Architecture Definition” practice area.)

An ineffective evaluation can result from the following:

- **the wrong people being involved in the evaluation:** If the architect is not involved in the evaluation, it is unlikely that enough information will be uncovered to make the evaluation worthwhile. Similarly, if the architecture’s stakeholders are not involved, the comprehensive goals and requirements for the architecture (against which it must be evaluated) will not emerge.
- **the evaluation taking place at the wrong time in the life cycle:** If the evaluation is too early, not enough decisions have been made, so there isn’t anything to evaluate. If the evaluation is too late, little can be changed as a result of it.
- **insufficient time for the evaluation:** If time is not planned for the evaluation, the people who need to be involved will not be able to give it their attention, the evaluation will not be conducted effectively, and the results will be superficial at best.
- **incorrect interpretation of the evaluation:** The results of any architecture evaluation should not be seen as a complete enumeration of all the risks in the development. Process deficiencies, resource inadequacies, personnel issues, and downstream implementation problems are risks that are unlikely to be exposed by an architecture evaluation.
- **the evaluation having the wrong focus:** The evaluation will not produce the desired results if the process does not cause the evaluation team to focus on the areas of the architecture that are problematic.
- **failure to follow up with a reevaluation:** As the architecture inevitably evolves, or the criteria for its suitability inevitably evolve, it should be reevaluated (perhaps using a lightweight version of the original evaluation) periodically to validate whether the organization is on the right track.

## Further Reading

[Clements 2001a]
Clements, Kazman, and Klein wrote a primer on software architecture evaluation that contains a detailed process model and practical guidance for applying the ATAM and compares it with other evaluation methods. Other methods, including ARID, are also covered.

[Del Rosso 2006a]
Del Rosso describes Nokia’s experience with evaluating architectures for product lines using a variety of evaluation methods.

[Parnas 2001a]
The work of Parnas and Weiss, in which they describe ADRs, remains the most comprehensive source of information on this approach.

[SEI 2007b]
The SEI’s SAT Web page provides publications about the ATAM and the SAAM, as well as other software architecture topics.

[Smith 1990a]
Smith’s work remains the definitive treatment of performance engineering.

[Smith 2001a]
The work of Smith and Williams is a good accompaniment to, but not a substitute, for Smith’s solo work [Smith 1990a].

[Zhao 1999a]
Zhao compiled a bibliography on software architecture analysis. His Web site, where the list is kept up-to-date, is cited on the SEI’s SAT Web page [SEI 2007b].

## Component Development

One of the tasks of the software architect is to produce the list of components that will populate the architecture. This list gives the development, mining, and acquisition teams their marching orders for supplying the parts that the software system will comprise. The term *component* is about as generic as the term *object*; definitions for each term abound. Simply stated, components are the units of software that go together to form whole systems (products), as dictated by the software architecture of the products. Szyperski offers a more precise definition that applies well [Szyperski 2002a]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

By *component development*, we mean the production of components that implement specific functionality within the context of a software architecture. The functionality is encapsulated and packaged and then integrated with other components using an interconnection method.

Software components trace their heritage back to the subroutine, which was the first unit of software reuse. Programmers discovered they could invoke a previously written segment of code and have access to its functionality while being blissfully unconcerned with its implementation, development history, storage management, and so forth. Soon, few people had to worry about how to code, say, a numerically stable double-precision cosine algorithm. Besides saving time, this practice elevated our thinking: we could think “cosine” but not have to think about storage registers and overflowing multiplications. It also elevated our languages: sophisticated subroutines were indistinguishable from primitive, atomic statements in the programming language.

What we now call component-based software development flows in an unbroken line from these early beginnings. Modern components, the latest instantiation of which are Web-based *services* that populate service-oriented architectures, are much larger and much more sophisticated, carry us much higher into domain-specific application realms, and have more complex interaction mechanisms than subroutine invocation (mechanisms that are standardized). However, the concepts and the reasons why we embrace them remain the same. In the same way that early subroutines liberated the programmer from thinking about details, component-based (or service-oriented) software development shifts the emphasis from *programming software* to *composing software systems*. Implementation has given way to integration as the focus. At its foundation is the assumption that sufficient commonality exists in many large software systems to justify developing reusable components to exploit and satisfy that commonality. Today, we look for components that provide large collections of related functionality all at once (instead of a cosine routine, think of *Mathematica*; instead of a relational database, think of an all-encompassing business platform; instead of an address book, think of Customer Resource Management systems) and whose interconnections with each other are loose and flexible. If we have control over the decomposition into components and the interfaces of each, the granularity and interconnection is determined by our system's software architecture. If the components are built externally, their granularity and interfaces are imposed on us, and they affect our software architecture.

The "Component Development" practice area is concerned with in-house development and how to build the components so that the instructions given to us in the architecture are carried out. New development should occur only after carrying out the activities in the "Make/Buy/Mine/Commission Analysis" practice area. Components obtained from sources other than new, in-house development are covered under the "Using Externally Available Software," "Mining Existing Assets," and "Developing an Acquisition Strategy" practice areas.

## Aspects Peculiar to Product Lines

For the purposes of product lines, components are the units of software that go together to form whole systems (products), as dictated by the product line architecture for the products and the product line as a whole. If we appeal to the Szyperski definition of components given above, "deployed independently" may simply mean installed into a product line's core asset base where they are made available for use in one or more products. "Third parties" are the product developers who assemble the component with others to create systems. The contractually specified interfaces are paramount, as they are in any software development paradigm with software architecture at its foundation.

The component development portion of a product line development effort focuses on providing the operational software that is needed by the products and that is to be developed in-house. Either the resultant components are included in the core asset base and hence used in multiple products in the product line or they are product-specific components. Components that are included in the core asset base must support the flexibility needed to satisfy the variation points specified in the product line architecture and/or the product line requirements. The needed functionality is defined in the context of the product line architecture. The architecture also defines where variation is needed.

The singular aspect of component development that is peculiar to product lines is providing the required variability in the developed components via the variation mechanisms described in the example practices for this practice area. These variation mechanisms must be chosen to adequately support the production strategy and production constraints (see Core Asset Development).

## Application to Core Asset Development

If a developed component is to be a core component, it must have an associated attached process that explains how any built-in component-level variation can be exercised in order to produce an instantiated version for a particular product. Developed components and their related artifacts (interface specifications, attached processes for instantiating built-in variability, test support, and so on) constitute a major portion of the product line's core asset base. Hand in hand with the software architecture that mandated them into existence, the core components form the conceptual basis for building products. Consequently, component development, as described above, is a large portion of the activity on the core asset development side of product line operations.

## Application to Product Development

If a developed component is not to be part of the core asset base, it is most likely specific to a particular product and therefore probably does not have much variability built into it. While the development task must obey the architecture as strictly as it must for core components, non-core development is likely to be simpler. Nevertheless, developers of non-core components would be wise to look for places where variability could be installed in the future, should the component in question ever turn out to be useful in a group of products. Components for a product are either (1) used directly from the core asset base, (2) used directly after binding the built-in variations, (3) used after modification or adaptation, or (4) developed anew. Since the first two cases are pro forma, we will discuss the last two.

**Adapting components:** Components that are being used in a context other than the one for which they were originally developed seldom fit their assigned roles exactly. There are techniques that can accommodate these differences. The *Adapter Design* pattern [Gamma 1995a], imposes an intermediary between two components. The adapter can compensate for mismatches in the number or types of parameters within a service signature, provide synchronization in a multithreaded interaction, and adjust for many other types of incompatibilities. Scripting languages can often be used to implement the adapter.

Another technique is to modify the component to fit its new environment. Doing so may be impossible if the source code is not available. And, even if it is possible, it is usually a bad idea. Cloning an existing component creates a new asset that must be managed and creates a dependency that cannot be expressed explicitly. It can vary independently of its parent component, making maintaining both pieces a difficult task. Object-oriented notations provide a semantic device to express this type of relationship by defining the dependent class in terms of an extension of the original class. Although similar devices do not exist at the component level, a new component may be implemented by deriving objects from those that implement the original component.

**Developing new components:** New development should occur only after a thorough search has been made of existing core assets. In some organizations, the product team may have to "contract" with a component development organization to build the needed component. If it is built by the product organization, product line standards should exist for creating the core assets that support the component.

Whether a product component is adapted or built from scratch, it should be reviewed ultimately for "promotion" to the core asset base (and, in fact, should be developed with that in mind). To help with that review, robustness analysis [Jacobson 1997a] can be applied to determine how flexible the product is with respect to future changes in requirements. By examining change cases (use cases that are not yet requirements), the team identifies system changes that are needed in order to support the new requirements and thereby provides a feedback loop to the component developers. Specifications for new components and modifications to existing ones are the outputs of this analysis.

## Example Practices

The example practices in this practice area all deal with component-level variation mechanisms. Architectural variation mechanisms are addressed in the "Architecture Definition" practice area.

**Variation mechanisms (1):** Jacobson, Griss, and Jonsson discuss the mechanisms for supporting variability in components (see Table 3) [Jacobson 1997a]. Each mechanism provides a different type of variability. The variation of functionality happens at different times depending on the type. Some of these variation types are included in the specification implicitly. For example, when a parameter is used, the specification includes the specific type of component mentioned in the contract or any component that is a specialization of that component. In the template instantiation example below, the parameter to the template is Container, which permits variation implicitly via the *Inheritance* pattern. The Container parameter can be replaced by any of its subclasses, such as Set or Bag.

One aspect of variability that is important in a product line effort is whether the variants must be identified at the time of product line architecture definition or can be discovered during the individual product's architectural phase. Inheritance allows for a variant to be created without the existing component having knowledge of the new variant. Likewise, template instantiation allows for the discovery of new parameter values after the template is designed; however, the new parameter must satisfy the assumptions of the template, which may not be stated explicitly in the interface of the formal parameter. In most cases, configuration further constrains the variation to a fixed set of attributes and a fixed set of values for each attribute.

*Table 3: Types of Variation [Jacobson 1997a]*

| Mechanism | Time of Specialization | Type of Variability |
|---|---|---|
| Inheritance | At class definition time | Specialization is done by modifying or adding to existing definitions.<br><br>Example: LongDistanceCall inherits from PhoneCall. |
| Extension | At requirements time | One use of a system can be defined by adding to the definition of another use.<br><br>Example: WithdrawalTransaction extends BasicTransaction. |
| Uses | At requirements time | One use of a system can be defined by including the functionality of another use.<br><br>Example: WithdrawalTransaction uses the Authentication use. |
| Configuration | Previous to runtime | A separate resource, such as file, is used to specialize the component.<br><br>Example: JavaBeans properties file |

| | | |
|---|---|---|
| Parameters | At component implementation time | A functional definition is written in terms of unbound elements that are supplied when actual use is made of the definition.<br><br>Example: calculatePriority(Rule) |
| Template instantiation | At component implementation time | A type specification is written in terms of unbound elements that are supplied when actual use is made of the specification.<br><br>Example: ExceptionHandler<Container> |
| Generation | Before or during runtime | A tool that produces definitions from user input.<br><br>Example: Configuration wizard |

**Variation mechanisms (2):** Anastasopoulos and Gacek expound on a somewhat different set of variation options that includes [Anastasopoulos 2000a]

- **aggregation/delegation:** an object-oriented technique in which the functionality of an object is extended by delegating the work it cannot normally perform to an object that can. The delegating object must have a repertoire of candidates (and their methods) and assumes a role resembling that of a service broker.
- **inheritance:** which assigns base functionality to a superclass and extended or specialized functionality to a subclass. Complex forms include dynamic and multiple inheritance, in addition to the more standard varieties.
- **parameterization:** as described above
- **overloading:** which means reusing a named functionality to operate on different data types. Overloading promotes code reuse but at the cost of understandability and code complexity.
- **properties in the Delphi language:** which are attributes of an object. Variability is achieved by modifying the attribute values or the actual set of attributes.
- **dynamic class loading in Java:** where classes are loaded into memory when needed. A product can query its context and that of its user to decide which classes to load at runtime.
- **static libraries:** which contain external functions that are linked to after compilation time. By changing the libraries, you can change the implementations of functions that have known names and signatures.
- **dynamic link libraries:** which give the flexibility of static libraries but defer the decision until runtime based on context and execution conditions
- **conditional compilation:** which puts multiple implementations of a module in the same file. One is chosen at compile time according to the appropriate preprocessor directives.
- **frame technology:** Frames are source files equipped with preprocessor-like directives that allow parent frames to copy and adapt child frames and form hierarchies. On top of each hierarchical assembly of frames lies a corresponding specification frame that collects code from the lower frames and provides the resulting ready-to-compile module.
- the ability of a program to manipulate data that represents information about itself or its execution environment or state. Reflective programs can adjust their behavior based on their context.
- **aspect-oriented programming:** which is described in the “Architecture Definition” practice area

- **design patterns:** which are extensible, object-oriented solution templates catalogued in various handbooks (for example, the work of Gamma and colleagues [Gamma 1995a]). We mentioned the Adapter Design pattern specifically as a variation mechanism earlier in this practice area.

### Practice Risks

The overriding risk in component development is building unsuitable components for the software product line applications. Doing so results in poor product quality, the inability to field products quickly, low customer satisfaction, and low organizational morale. Unsuitable components can come about by

- **an ill-conceived component development effort:** A major risk in component development is doing it prematurely or doing it at all, when another option would have been better. For example, building a component before an architecture is in place can cause interface and integration problems. Building a component when a better option would be to buy, mine, or commission it will lead to increased cost and complexity. Building a component especially for a product instead of using an available core asset will lead to the disintegration of the product line.
- **insufficient variability:** Components not only must meet their behavioral and quality requirements (as imposed on them by the product line's software architecture) but also must be tailorable in preplanned ways to enable product developers to instantiate them quickly and reliably in the correct forms for specific products.
- **too much variability:** Building in too much variability can prevent the components from being understood well enough to be used effectively or can cause unforeseen errors when the variabilities conflict with each other.
- **choosing the wrong variation mechanism(s) for the job:** The wrong choice can result in components that cannot be tailored when necessary.
- **poor-quality components:** While any poor-quality components will set back a software development effort, those that are core asset components will undermine the entire product line. Product builders will lose confidence in the core asset builders, and pressure will mount to bypass them. Applying practices from the "Testing" practice area can ameliorate this risk.

### Further Reading

[Jacobson 1997a] & [Anastasopoulos 2000a]
Together, these two works provide a superb compendium of component-level variation mechanisms that are available to a product line component developer.

[Szyperski 2002a]
Szyperski provides a comprehensive presentation on components that gives a survey of component models and covers supporting topics such as domain analysis and component frameworks.

## Mining Existing Assets

Mining existing assets refers to resurrecting and rehabilitating a piece of an old system to serve in a new system for which it was not originally intended. Often it simply refers to finding useful legacy code in an organization’s existing systems portfolio and reusing it within a new application. However, the code-only view completely misses the big picture. We have known for years that in the grand scheme of things, code plays a small role in the cost of a system, because it’s not the hard part of system/software development. Rich candidates for mining include a wide range of assets besides code—assets that will pay lucrative dividends. Business models, rule bases, requirements specifications, schedules, budgets, test plans, test cases, coding standards, algorithms, process definitions, performance models, and the like are all wonderful assets for reuse. The only reason so-called “code reuse” pays at all is because of the designs, algorithms, and interfaces that come along with the code [Clements 2002c, p. 99].

For example, whole or partial architectures and the design decisions they embody (captured by the documented rationale) are especially valuable. If a mined architecture is suitable, the components that originally populated it can most likely be migrated along with it. But to determine the fitness for reuse of either the architecture or its components, you must have a thorough architectural understanding of the legacy system. And, of course, the architect may be long gone. If good documentation does not exist, you might need to reconstruct the architecture to reveal the interactions and relations among the architecture’s components. Reconstruction will illuminate constraints for how the components, if mined, can interact within the architecture of the new or updated software. It can also help you understand the tradeoff options available for reusing components in a new or improved way [Kazman 2002a, O’Brien 2002a]. Once the architecture has been extracted, it can be evaluated for suitability using the techniques described in the “Architecture Evaluation” practice area.

Documentation is an asset that is often overlooked and may have significant reuse potential. Much of the corporate knowledge about the software assets may be captured in existing legacy documentation, which makes them highly desirable candidates for mining and rehabilitation. That is especially true when the associated software assets are being mined and rehabilitated and they closely correlate with one another.

Mining involves understanding what is available, what is needed, and how rehabilitation works. That understanding requires support from analysts who are familiar with both the legacy system and the new system. For software assets, rehabilitation usually requires the support of the new system’s architect, who will direct how the assets will be integrated into the new architecture.

For software assets, focus first on large-grained assets that can be wrapped or that will require only interface changes rather than changes in large chunks of their underlying algorithms or code. Determine how the candidate asset can fit into the architecture of the targeted new system. Don’t forget to consider the requirements for performance, modifiability, reliability, and other nonbehavioral qualities. In addition, be sure to include all the non-software assets associated with the software: requirements, design, test, and management artifacts.

Once the existing assets have been organized and understood and candidate assets for mining have been identified, the rehabilitation of these assets can begin. In many ways, a mining initiative that involves the extensive rehabilitation of assets can resemble a reengineering project [Seacord 2003a, Sneed 2001a, Ulrich 2002a] or a development project in its own right. Technical planning (as described in the “Technical Planning” practice area) can help in planning and coordinating the effort.

Normally, mining refers to legacy assets previously developed by the organization doing the mining. However, the rehabilitation activities associated with mining also apply to externally available software such as open source software that has entered the organization from the outside.

### Aspects Peculiar to Product Lines

Mined assets for a product line must have the same qualities as newly developed core assets. Mined assets must be (re)packaged with reuse in mind and meet the product line requirements. Accordingly, the mined assets must align with the product line architecture and meet the quality goals consistent with those of the product line. Product lines must focus on the strategic, large-grained reuse of the mined assets. The primary issues that motivate large-scale reuse for a product line are schedule, cost, and quality. The mined and rehabilitated assets must meet the needs of the plurality of systems in the product line. Since a product line accommodates a longer and wider view of future system change, any mined asset must be robust enough to accommodate such change gracefully.

When mining an asset (software or otherwise) for a software product line, an analyst should consider

- its alignment with the requirements for immediate products in terms of both common features and variation points
- its appropriateness for potential future products
- whether it will be used as a core asset or for a specific product
- the amount of effort required to make its interface conform to the constraints of the product line architecture
- its extensibility with respect to its potential future (based on the architecture’s expected evolution)
- its maintenance history
- other assets (for example, script and data files) that may be required from the legacy system
- its projected long-term cost

When mining software assets for single systems, we look for components that perform specific functions well. However, for product line systems, quality attributes such as maintainability and suitability become more important over time. Thus, we might accept mined assets for product lines that are suboptimal in fulfilling specific tasks if they meet the product line’s critical quality attribute goals. An asset’s total cost of ownership across the products for which it will be used should be lower than the sum of similar assets mined for one-time use.

### Application to Core Asset Development

The process of mining existing assets is largely about finding suitable candidates to be core assets of the product line. Software assets that are well structured and well documented and have been used effectively over long periods of time can sometimes be included as product line core assets with little or no change. Software assets that can be wrapped to satisfy new interoperability requirements are also desirable. Assets that don’t satisfy these requirements are undesirable and may have higher maintenance costs over the long term. Depending on the legacy inventory and its quality, an assortment of candidate assets is possible, from architectures to small pieces of code.

An existing architecture should be analyzed carefully before being accepted as the pivotal core asset—the product line architecture. See the “Architecture Evaluation” practice area for a discussion of what that analysis should entail.

Candidate software assets must align with the product line architecture, meet specified component behavior requirements, and accommodate any specified variation points. In some cases, a mined component may represent a potentially valuable core asset but won’t fit directly into the product line architecture. Usually, the component will need to be changed to accommodate the constraints of the architecture. Sometimes a change in the architecture might be easier, but it will have implications for other components, for the satisfaction of quality goals, and for the support of the products in the product line.

Once in the core asset base for the product line, mined assets are treated in the same way as newly developed assets. Some assets, though, that may have been mined for expediency in fielding a new product, may be less than robust and be earmarked for future replacement via new development.

### Application to Product Development

Although it is reasonable to use mined assets for components that are unique to a single product in the product line, doing so will make the mining activity indistinguishable from mining in the non-product-line case. The same issues discussed above (paying attention to quality attributes, architecture, cost, and time to market) will still apply. And it will be worth taking a long, hard look at whether the mined component really is unique to a single product or could be used in other products as well, thus making the cost of its rehabilitation more palatable. In that case, the team responsible for mining would be wise to look for places where variability could be installed in the future, should the asset in question ever turn out to be useful in a group of products.

### Example Practices

**SEI Options Analysis for Reengineering (OAR):** OAR is a method that can be used to evaluate the feasibility and economy of mining existing components for a product line. OAR operates like a funnel in which a large set of potential assets is screened out so that the effort can focus on a smaller set that will most effectively meet the technical and programmatic needs of the product line. OAR prescribes the following steps [Bergey 2001a, Bergey 2002a, Bergey 2003a]:

1. **Establish the mining context:** First, capture your organization’s product line approach, legacy base, and expectations for mining components. Establish the programmatic and technical drivers for the effort, catalogue the documentation available from the legacy systems, and identify a broad set of candidate components for mining. This task establishes the needs of the mining effort and begins to illuminate the types of assets that will be most relevant for mining. It also identifies the available documentation and artifacts and enables focused efforts to close any gaps in the existing documentation.
2. **Inventory components:** Next, identify the legacy system components that can potentially be mined for use in a core asset base for the product line. During this activity, identify the required characteristics of the components (such as functionality, language, infrastructure support, and in-

terfaces) in the context of the product line architecture. This activity creates an inventory of candidate legacy components together with a list of the relevant characteristics of those components. It also creates a list of needs that cannot be satisfied through the mining effort.

3. **Analyze candidate components:** Next, analyze the candidate set of legacy components in more detail to evaluate their potential use as product line components. Screen them on the basis of how well they match the required characteristics. This activity provides a list of candidate components, together with estimates of the cost of rehabilitating those components and the effort required.
4. **Analyze mining options:** Then, assemble different aggregations of components and analyze the feasibility and viability of mining them based on their cost, effort, and risk.
5. **Select a mining option:** Finally, select the mining option that can best satisfy the organization's mining goals by balancing the programmatic and technical considerations of the organization. To do that, establish drivers for making a final decision, such as cost, schedule, risks, and difficulty. (This activity might also help you determine the tradeoffs.) Evaluate each mining option (component aggregation) on the basis of how well it satisfies the most critical driver. Then after you select an option, write a final report to communicate the results of the OAR process.

OAR has been used to make decisions on mining components for a satellite-tracking system [Bergey 2001a]. The process has also been used to evaluate (1) the extent to which components proposed by suppliers for reuse in a product line would meet the product line's stated needs and (2) the types of changes that would be required to fit the component into the product line [Bergey 2003a, Muller 2003a]. OAR is in the process of being extended to handle other asset types such as unit test cases and documentation.

**Architecture recovery/reconstruction tools:** Some tools that are available to assist in the architecture reconstruction process include Rigi [Muller 1988a], the Software Bookshelf [Finnegan 1997a], DISCOVER [Tilley 1998a], the Dali workbench [Kazman 1998a], and the SEI ARMIN tool [O'Brien 2003a].

The ARMIN tool is a flexible, lightweight tool for architecture reconstruction. It uses information extracted by other tools to generate architectural views. Using ARMIN involves five steps:

1. **information extraction:** which uses tools such as parsers to extract information from existing design and implementation artifacts such as the source code
2. **database construction:** which stores the extracted information in a database for future analysis. This step may involve changing the format of the data.
3. **view fusion:** which augments the extracted information by combining information to generate a set of low-level views of the software
4. **architecture view composition:** which generates a set of architecture views through abstraction, visualizes them, and then enables the user to explore and manipulate them
5. **architecture analysis:** which evaluates the resultant architecture and, in some cases, evaluates the conformance of the as-built architecture obtained from reconstruction to an as-designed architecture

Tool support makes mining undocumented software assets more effective and significantly less cumbersome by reducing the time it takes to ascertain what a piece of software does and how it interacts with other parts of the system. Tools can be brought to bear that automatically chart interconnections of various kinds among software elements. More valuable than tools, however, are the people who worked on and are knowledgeable about the legacy software. Find them if you can. They can tell you the strengths and weaknesses of the software that weren't written down, and they can give you the "inside story" that no tool can hope to recover.

**Mining architectures:** In some cases, the software architecture of an existing system can become the product line architecture. The SEI Mining Architectures for Product Lines (MAP) method determine whether the architectures of existing systems are similar and whether the corresponding systems have the potential of becoming a software product line [O'Brien 2001a]. The MAP method combines techniques for architecture reconstruction and product line analysis to analyze the architectural patterns and attributes of a set of systems. This analysis determines whether the systems have similar components and connections between their components and examines their commonalities and variabilities. MAP has been used in the development of a prototype product line architecture for a sunroof system. MAP and OAR can also be used together effectively: MAP supports decision making for reusing architectures, while OAR supports decision making for identifying components that fit within the constraints of the architecture.

**Requirements reuse and feature interaction management:** Developers realize that complex applications are often built best by using a number of different components, each performing a specialized set of services. However, those components, each embodying different requirements in different service domains, can interact in unpredictable ways. For this reason, designing components to minimize or at least manage interaction is an issue. This problem of interaction becomes even more significant when reusing requirements. Interactions must be detected and resolved in the absence of a specific implementation framework. Shehata, Eberlein, and Hoover stress that understanding interaction management is key to understanding how to reuse requirements. They describe a conceptual process framework for formulating and reusing requirements [Shehata 2002a]. Reusable requirements are classified into three levels of abstraction for software requirements: domain-specific requirements, generic requirements, and domain-requirements frameworks. This classification is used as the basis for a reusability plan that underscores the importance of interaction management.

**Wrapping:** Wrapping involves changing the interface of a component to comply with a new architecture without changing the component's other internals. In fact, pure wrapping involves no change whatsoever in the component; rather, a new thin layer of software is interposed between the original component and its clients. That layer provides the new interface by translating to and from the old. There are enormous advantages to reusing existing assets with little or no internal modification through wrapping. As soon as any modification takes place, the associated documentation changes, the test cases change, and a ripple effect takes place that influences other associated software. Wrapping prevents that ripple effect and allows the "as-is" reuse of many of the assets associated with the software component, such as its test cases and internal design documentation. The idea is to translate the "as-is" interface to the "to-be" interface. Weiderman and colleagues discuss some of the available wrapping techniques [Weiderman 1997a]. Seacord and colleagues discuss a case study that applied several wrapping techniques [Seacord 2001a].

**Adapting components:** Software components that are being used in a context other than the one for which they were originally developed often do not exactly fit their assigned roles. One technique that can accommodate these differences is use of the *Adapter Design* pattern [Gamma 1995a]. Using that pattern imposes an intermediary between two components. The adapter can compensate for mismatches in the number or types of parameters within a service signature, provide synchronization in a multithreaded interaction, and adjust for many other types of incompatibilities. Scripting languages can often be used to implement the adapter.

**Domain-specific reuse:** An approach by Ganesan and Knodel looks at domain-specific reuse [Ganesan 2005a]. The goal of their approach is to reduce the amount of source code that a human expert has to explore to identify domain-specific software components. They outline a 10-step process for component extraction. The basic premise of this approach is that reusable classes have certain quality attributes—like functional usefulness, readability, testability, and so forth—that are mapped onto metrics. The approach classifies the domain-specific classes based on the metrics derived. An expert has to validate only a small number of proposed candidates that, if accepted, become the foundations for the reusable components.

Analyzing product line adequacy: The Fraunhofer Product Line Software Engineering (PuLSE) and Fraunhofer Architecture- and Domain-Oriented Reengineering (ADORE) are methods that can be used to analyze existing components to determine their adequacy for use within a product line. Knodel and Muthig report on the application of these methods in an industrial case study [Knodel 2005a]. In that study, several techniques were used to answer a set of questions about the existing components that determine their adequacy. The techniques applied included static architecture evaluation, variability analysis, clone detection, metric computation, and naming and decomposition analysis, as well as review of code comments.

## Practice Risks

The major risks associated with mining are (1) failure to find the right assets and (2) choosing the wrong assets. Both will result in schedule slippage and wasted staff time. A secondary risk is inadequate support for the mining operation, which will result in a failed operation and the (misguided) impression that mining is not a viable option.

Specific risks associated with mining operations include

- **a flawed search:** The search for reusable assets may be fruitless, resulting in a waste of time and resources. Or, relevant assets may be overlooked, resulting in a waste of time and resources spent duplicating what already exists. A special case of the latter is when noncode assets are shortsightedly ignored. To minimize both of these risks, build a catalogue of your reusable assets (including noncode assets) and treat it as a core asset of the product line. Doing so will save time and effort next time.
- **an overly successful search:** There may be too many similar assets, resulting in too much effort spent on analysis.

- **fuzzy criteria:** The criteria for what to search for should be specific enough to avoid an overly successful search, yet be general enough to include all the viable candidates.
- **failure to search for non-software assets:** Failure to consider non-software assets in your search-such as specifications, test suites, procedures, budgets, work plans, requirements, and design rationale-will reduce the overall effectiveness of any mining operation.
- **inappropriate assets:** Assets recovered from a search may appear, at first, to be usable but later turn out to be of inferior quality or unable to accommodate the scope of variation required.
- **bad rehabilitation estimates:** Initial estimates of the cost of rehabilitation may be inadequate, leading to escalating and unpredictable costs.

Organizational issues leading to mining risks include

- **a lack of corporate memory:** Corporate memory may not be able to provide sufficient data to analyze or use the software asset effectively.
- **inappropriate methods:** The wrong reengineering methods and tools may be selected, leading to flawed results and schedule and cost overruns.
- **a lack of tools:** Tools required for the mining effort may not be integrated to the extent necessary, leading to risky and expensive workarounds.
- **turf conflicts:** Potential turf conflicts may undermine the decision process when selecting among similar candidate assets. Or, a repository of assets may be off-limits for political or organizational reasons.
- **an inability to tap the needed resources:** The organization might be unable to free the resources needed to rehabilitate or renovate the asset. Those resources must be freed from the group that originally created the asset.

## Further Reading

[Ganesan 2005a]
This paper outlines a 10-step process for component extraction that classifies a set of components based on determining a set of metrics for the classes that underlie those components.

[Knodel 2005a]
This paper outlines a case study in applying a variety of techniques to determine the adequacy of components for use within a product line. Several techniques were used to answer a set of questions about the components that determine their adequacy.

[Seacord 2003a]
This book on modernizing legacy systems by Seacord, Plakosh, and Lewis provides guidance on how to implement a successful modernization strategy and specifically describes a risk-managed, incremental approach that encompasses changes in software technologies, engineering processes, and business practices.

## Requirements Engineering

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements . . . No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later" [Brooks 1987a].

So wrote Fred Brooks in 1987, and so it remains today [Davis 1990a, Faulk 1997a]. "The inability to produce complete, correct, and unambiguous software requirements is still considered the major cause of software failure today" [Dorfman 1997a].

Requirements are statements of what the system must do, how it must behave, the properties it must exhibit, the qualities it must possess, and the constraints that the system and its development must satisfy. The Institute of Electrical and Electronics Engineers (IEEE) defines a requirement as

1. a condition or capability needed by a user to solve a problem or achieve an objective
2. a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document
3. a documented representation of a condition or capability as in definition 1 or 2 [IEEE 1990a]

Requirements engineering emphasizes the use of systematic and repeatable techniques that ensure the completeness, consistency, and relevance of the system requirements [Sommerville 1997a]. Specifically, requirements engineering encompasses requirements elicitation, analysis, specification, verification, and management, where

- **Requirements elicitation** is the process of discovering, reviewing, documenting, and understanding the user's needs and constraints for the system.
- **Requirements analysis** is the process of refining the user's needs and constraints.
- **Requirements specification** is the process of documenting the user's needs and constraints clearly and precisely.
- **Requirements verification** is the process of ensuring that the system requirements are complete, correct, consistent, and clear.
- **Requirements management** is the process of scheduling, coordinating, and documenting the requirements engineering activities (that is, elicitation, analysis, specification, and verification) [Dorfman 1997a].

Requirements engineering is complex because of the three roles involved in producing even a single requirement: the requestor (referred to as the "user" in the IEEE definition), the developer (who will design and implement the system), and the author (who will document the requirements). Typically, the requestor understands the problem to be solved by the system but not how to develop a system. The developer understands the tools and techniques required to construct and maintain a system but not the problem to be solved by that system. The author needs to create a statement that communicates unambiguously to the developer what the requestor desires. Hence, requirements address a fundamental communications problem.

The communications problem is further compounded by the number and diversity of system requestors. In practice, any system stakeholder has needs and expectations (that is, requirements) for the system. The role of system stakeholder is played by any of the various people or systems involved in or affected by a system's development. This group includes executives (who know the organization's business goals and constraints), end users (who know how the products will be used), marketers (who know the market demands), technical managers (who know the available personnel), and developers (who know the available tools and technology). It can potentially include legal experts, government agencies, and insurance experts. Successful requirements engineering depends on the identification and solicitation of the appropriate community of stakeholders.

Reconciling the diverse needs and expectations of the various system stakeholders necessitates tradeoffs—that is, decisions between potentially conflicting requirements from different stakeholders. Tradeoffs require mechanisms for capturing and analyzing the different stakeholder requirements, for recognizing the conflicting requirements of different stakeholders, for deciding among those conflicting requirements, and for recording the results of those decisions. Tradeoffs are captured as system decisions that are linked to the affected requirements.

Requirements are pervasive, continuously affecting all development and maintenance phases of a system's development by providing the primary information needed during them. The requirements form a trigger mechanism for the development and maintenance efforts. Testing, for instance, depends on a precise statement of quality and behavioral requirements to define the standard of correctness against which to test.

The longer the system's lifetime, the more it is exposed to changes in the requirements that result from changes in the needs and concerns of its stakeholders. For example, the end user's demands can change as a result of new features offered by a competing organization's products. The organization's business goals and constraints can change as a result of market demands, new laws, or new insurance regulations. New technologies and software tools such as operating systems can change the way the system is constructed. Such changes require mechanisms for managing them—that is, a requirements-change-management process. That process is based on the traceability links between the requirements and the system work products and between the requirements and the related decisions and tradeoffs.

The following sections describe requirements engineering for products: Core Asset Development describes other kinds of requirements that also need to be managed for a software product line, such as the production constraints.

## Aspects Peculiar to Product Lines

Product line requirements constitute an important and tangible core asset. They define the products in the product line together with the features of and the constraints on those products. These requirements are tightly coupled with the product line's scope definition and evolve together [Clements 2002c, p. 180]. Requirements common across the product line are written with variation points that can be filled in or exercised to create product-specific requirements. These variation points may be small, such as replacing a symbolic name such as "max_nbr_of_users" with a value such as "150," or may be substantial, such as leaving a placeholder for an entire specification of part of the behavior.

Sometimes the variation point will map to null for a product, corresponding to a feature that the product does not have. And there must be some mechanism by which the complete set of requirements for a particular product (common plus unique) can be produced quickly and easily, implying that the product-specific requirements are stored as a set of deltas relative to the product-line-wide requirements specification.

Requirements engineering for a product line differs from requirements engineering for single systems as follows:

- **Requirements elicitation** for a product line must capture anticipated variations explicitly over the foreseeable lifetime of the product line. This means that the community of stakeholders is probably larger than for single-system requirements elicitation and may well include domain experts, market experts, and others. Requirements elicitation focuses on the scope, explicitly capturing the anticipated variation by the application of domain analysis techniques, the incorporation of existing domain analysis models, and the incorporation of use cases that capture the variations that are expected to occur over the lifetime of the product line (e.g., change cases [Ecklund 1996a] and use case variation points [Jacobson 1997a]).
- **Requirements analysis** for a product line involves finding commonalities and identifying variations. It may also involve a more vigorous feedback mechanism to the requestors, pointing out where a particular system might achieve economies if it could use more common requirements. And requirements analysis has the product line scope as one of its inputs—an artifact that does not exist outside the product line context. Requirements analysis includes a commonality and variability analysis (a technique used frequently in domain analysis) on the elicited product line requirements to identify the opportunities for large-grained reuse within the product line. Two such techniques are feature-oriented domain analysis (FODA) [Kang 1990a] and use cases [Jacobson 1997a].
- **Requirements specification** now includes documentation of a product-line-wide set of requirements and product-specific requirements. The product-line-wide requirements include symbolic placeholders that the various product-specific requirements documents fill in, expand, or instantiate.
- **Requirements verification** now includes a broader reviewer pool and occurs in stages. First, the common product line requirements must be verified. Later, as each product comes on the scene (or is updated), its product-specific requirements must be verified. But the product-line-wide requirements must also be verified to make sure that they make sense for this product.
- **Requirements management** must now make allowances for the dual nature of the requirements engineering process and the staged (common, specific) nature of the activity. Change-management policies must provide a formal mechanism for proposing changes in the product line and supporting the systematic assessment of how the proposed changes will impact the product line. Change-management policies govern how changes in the product line requirements are proposed, analyzed, and reviewed. The coupling between the product line requirements and the core assets is leveraged by the use of traceability links between those requirements and their associated core assets. Changes in the requirements can then trigger the appropriate changes in the related core assets [Sommerville 1997a]. The "Configuration Management" practice area describes change management in more detail.

### Application to Core Asset Development

First of all, the requirements artifacts produced by requirements engineering are important core assets in their own right. Beyond that, requirements engineering creates the product line requirements that feed the development and acquisition of other core assets. The requirements artifacts will help to

- determine the feasibility and refine the scope and business case of a product line. The initial version of the business case is frequently based on an informal notion of the scope. Requirements engineering refines the scope, and hence the business case, by determining more precisely the requirements for the product line. That more precise definition of the product line provides input to the business case activity for a correspondingly more precise determination of that product line's feasibility.
- lay the groundwork for the product line architecture, which must accommodate the commonalities and variabilities identified in the requirements
- ensure that the other core assets support the anticipated variations
- determine the schedules and budgets for the product line and its products
- create the test cases and other test artifacts for products in the product line

A significant difference in requirements engineering for product lines involves a rapid, initial pass through the requirements for key stakeholders to initiate early design work, capturing the high-level requirements that affect the initial design (that is, the architecturally significant requirements [Jacobson 1997a]). The purpose of this pass is to compress the time to initial delivery and to demonstrate the feasibility of and establish the credibility of the product line approach—in short, to provide an early justification of the investment [Graham 1998a]. Although this early justification is desirable in single-product development, it is more critical for a software product line.

### Application to Product Development

Requirements engineering plays a key role in

- determining the feasibility of producing a particular product as part of the product line. You can use a statement of the requirements specific to that candidate product to help estimate the cost of developing the product.
- the production, testing, and deployment of the particular product. Requirements play a role in these activities just as they do for single-system development.
- the evolution of the product line (that is, the incorporation of changes) that results from that product development. Product-specific requirements often "grow up" to become product line requirements if they can be slightly generalized or if they pop up in more than one product. That is the primary mechanism for the evolution of software product lines over time.

To expand on the first point, determining the feasibility of an additional product in a product line is an ongoing activity that is part of building the business case for that product. The initial version of that business case is frequently based on an informal description of the prospective product. Requirements engineering activities—particularly elicitation and analysis—support the business case for the product by determining more precisely the requirements for that product. The product line requirements guide the elicitation of the specific requirements for that product. Requirements analysis determines the variance between the product line and the product-specific requirements. That variance provides input to the business case activity for a correspondingly more precise cost estimate for building the specific product as part of the product line (that is, determination of that product's feasibility).

## Example Practices

**Domain analysis techniques:** These techniques can be used to expand the scope of the requirements elicitation, to identify and plan for anticipated changes, to determine fundamental commonalities and variations in the products of the product line, and to support the creation of robust architectures. (See the "Understanding Relevant Domains" practice area.) One of these techniques, FODA, has been incorporated recently in a newer approach to requirements engineering for product lines. Product Line Analysis (PLA) combines traditional object-based analysis with FODA feature modeling and stakeholder-view modeling to elicit, analyze, specify, and verify the requirements for a product line. Feature modeling facilitates the identification and analysis of the product line's commonality and variability and provides a natural vehicle for requirements specification. Stakeholder-view modeling supports the completeness of the requirements elicitation, while the PLA work products (the object model, use case model, feature model, and product line terminology dictionary) provide incremental verification of the requirements modeling [SEI 2007d, Chastek 2001a].

**Stakeholder-view modeling:** This technique can be used to support the prioritized modeling of the significant stakeholder requirements for the product line. Viewpoint modeling is based on the recognition that a system must satisfy the needs and expectations of the many system stakeholders, who all have their own perspectives (views) of the system. Each stakeholder view can be modeled separately as a set of system requirements. These models are core assets that support the explicit identification of conflicts and determination of tradeoffs, among the needs of the system stakeholders [Sommerville 1997a].

**Feature modeling:** This technique can be used to complement object and use case modeling and to organize the results of the commonality and variability analysis in preparation for reuse. Features are user-visible aspects or characteristics of a system that are organized into a tree of And/Or nodes to identify the commonalities and variabilities within the system. Feature modeling is an integral part of the FODA method [Kang 1990a] and the feature-oriented reuse method (FORM) [Kang 1998a]. In the latter, the requirements for a family of related systems are organized according to that family's features. The commonalities and variabilities within those features are then exploited to create a set of reference models (that is, software architectures and components) that can be used to implement the products of that family. Feature modeling has also been integrated into the reuse-driven software engineering business (RSEB) [Jacobson 1997a, Griss 1998a].

**Use case modeling:** This technique can be used with variation points to capture and describe commonality and variability within product line requirements. A *variation point* is a location within a use case where a variation (that is, variability) occurs. That variation is captured in a variant that describes the context and mode of the variation. The mechanisms supported for capturing and describing different types of variation within use cases include inheritance, uses, extensions, extension points, and parameterization [Jacobson 1997a].

**Change-case modeling:** This technique can be used to explicitly identify and capture anticipated changes in a system and to ultimately incorporate them explicitly in the design to enhance its long-term robustness. Change cases are use cases that describe potential future requirements for a system. They are linked to the existing system use cases that will be affected if and when the future requirements are adopted. The inclusion of change cases allows the designers to plan for and more effectively accommodate anticipated changes [Ecklund 1996a].

**Traceability of requirements to their associated work products:** This technique can be used to ensure that the design and implementation of a system satisfy the requirements for that system. Requirements traceability links the requirements backward to their sources (a stakeholder, for example) and forward to the resulting system development work products (a component, for example). In addition to assisting with the elicitation and verification of requirements, requirements traceability is critical in determining the potential impact of proposed changes in a system [Ramesh 1997a, Sommerville 1997a].

## Practice Risks

The major risk associated with requirements engineering is failure to capture the right requirements over the life of the product line. Documenting the wrong or inappropriate requirements, failing to keep the requirements up-to-date, or failing to document the requirements at all puts the architect and the component developers at a grave disadvantage. They will be unable to produce systems that satisfy the customers and fulfill the market expectations. Inappropriate requirements can result from the following:

- **failure to distinguish between product-line-wide requirements and product-specific requirements:** These different kinds of requirements have different audiences in a product line. The core asset builders need to know the requirements they must build to, while the product-specific software builders must know what is expected of them.
- **insufficient generality:** Insufficient generality in the requirements leads to a design that is too brittle to deal with the change actually experienced over the lifetime of the product line.
- **excessive generality:** Overly general requirements lead to excessive effort in producing both core assets (to provide that generality) and specific products (which must turn that generality into a specific instantiation).
- **wrong variation points:** Incorrect determination of the variation points results in inflexible products and the inability to respond rapidly to customer needs and market shifts.
- **failure to account for qualities other than behavior:** Product line requirements (and software requirements in general) should capture requirements for quality attributes such as performance, reliability, and security.

In addition, requirements engineering for a product line is subject to the risks enumerated in the "Understanding Relevant Domains" practice area.

### Further Reading

[Birk 2003a]
This Fraunhofer Institute report is based on the findings of the Gesellschaft für Informattik (GI) Working Group, which includes organizations such as Hewlett-Packard, Bosch, RWTH Aachen, and sd&m. Birk and colleagues provide a good description of the problems associated with requirements engineering for software product lines from the industry point of view.

[Davis 1990a] & [Sommerville 1997a]
These are two good textbooks for requirements engineering. Of particular value in Somerville and Sawyer's book is a table that describes the basic, intermediate, and advanced requirements engineering guidelines.

[Dorfman 1997a]
This is a tutorial in book form that is based on the most significant requirements engineering papers of the last two decades.

[Faulk 1997a]
Faulk provides an excellent, brief introduction to requirements engineering, describing such techniques as functional decomposition, structured analysis, operational specification, and object-oriented analysis.

[Schmid 2006a]
A continuing problem with the adoption of software product lines has been the lack of effective tool support for product line requirements engineering. This report describes the Requirements Management for Product Lines (REMAP) software tool. REMAP is a prototype that operates on top of existing requirements engineering tools, such as DOORS, and provides the additional product line support lacking in those existing commercial tools. The earlier work of Schmid and colleagues provides a more complete description of REMAP [Schmid 2005a].

## Software System Integration

Software system integration refers to the practice of combining individually tested software components into an integrated whole. Software is integrated when components are combined into subsystems or when subsystems are combined into products. Components may be integrated after all are implemented and tested as in a waterfall model or a "big bang" approach. In either, software system integration appears as a discrete step toward the end of the development life cycle between component development and integration testing. Continuous integration is a much less risky approach wherein the components and subsystems are integrated as they are developed into multiple working mini-versions of the system. Object technologists were early proponents of incremental development, and object-oriented development methods, such as the Unified Process [Alhir 2002a], are based on the principle of ongoing integration practices.

Continuous integration offers several advantages over a waterfall, "big bang" approach. Developers split a product into several builds, or partial products, that can be integrated individually. These builds may be chunked into "vertical" increments, covering subsystems, or increments may cross subsystem boundaries to produce a partial end-to-end product. In both cases, integration is iterative in that a sequence of incremental builds yields results successively closer to the desired product. Customers gain an advantage in seeing a working product early in the development, and developers can verify performance or other quality factors in a working environment, rather than relying on models or simulations. Continuous integration also decreases integration risk, because developers can identify integration problems, often the most complex to address, during early iterations of software integration.

Integration is bound up in the concept of component interfaces. Recall from the "Architecture Definition" practice area that an interface between two components is the set of assumptions that the programmers of each component can safely make about the other component [Parnas 1972a]. These assumptions include the component's behavior, the resources it consumes, how it acts in the face of an error, and other assumptions that should be documented as part of a component's interface. This definition is in stark contrast to the simplistic (and quite insufficient) notion of *interface* that merely refers to the "signature" or syntactic interface that includes only the component's name and parameter types. This definition of *interface* may let two components compile together successfully, but only the Parnas definition (which subsumes the simpler one) defines how two components work together correctly. When interfaces are defined thoughtfully and documented carefully, integration proceeds much more smoothly, because they define how the components will connect to and work with each other.

## Aspects Peculiar to Product Lines

In a product line effort, we identify two stages of software system integration:

1. **Core assets are integrated into the core asset base as part of core asset development, including review or test:** This merging may take the form of "pre-integration" and cover more than software components. As assets are developed and reviewed or tested, they migrate to the asset base. Under pre-integration—a continuous integration approach—the assets may be used to support other asset development or partial product development. The production strategy—the overall approach for realizing core assets—guides this activity.
2. **Core assets are integrated during the building of an individual product:** Here, the production plan guides the process, defining how developers will select the appropriate core assets from all the available ones and construct a product. All assets such as requirements, architecture, processes, and testing assets contribute to the construction and must be integrated into a delivered product. Integration may involve tailoring assets according to planned variations or developing product-unique software components, requirements, tests, and so forth.

For the core asset base, pre-integrating as many of the software core assets as you can makes product building a much more economical operation [Clements 2002c, p. 118]. This pre-integration can yield a "virtual" or test product that mirrors an actual, deliverable product. In both core asset base and product integration, you need to consider integration early on in the development of the production plan and architecture for the entire product line. The goal is to make software system integration straightforward and predictable.

In a product line, the effort involved in software system integration lies along a spectrum. At one end, the effort is almost zero. If you know all the products' potential variations in advance, you can produce an integrated parameterized template of a generic system with formal parameters. You can then generate final products by supplying the actual parameters specific to the individual product requirements and launching the construction tool (along the lines of the UNIX Make utility). In this case, each product consists entirely of core components; no product-specific code exists. This is the “system generation” end of the integration spectrum.

At the other end of the spectrum, considerable coding may be involved to bring the right core components together into a cohesive whole. Perhaps the components need to be wrapped, or perhaps new components need to be designed and implemented especially for the product. In these situations, the integration more closely resembles that of a single-system project.

Most software product lines occupy a middle point on the spectrum. Obviously, the closer to the generation side of the spectrum you can align your production approach, the easier integration will be and the more products you will be able to turn out in a short period of time.

However, circumstances may prevent you from achieving pure generation. Perhaps a new product has features you didn’t consider when developing the asset base, your application area prevents you from knowing all the variations up front, or the variations are so numerous or complex or interact with each other in such complicated ways that building the construction tool will be too expensive. And perhaps you prefer to produce fewer products over a long time period rather than turn out many products in a short amount of time. In that case, the construction tool may be less appealing.

In software system integration for product lines, the cost of integration is amortized over many products. Once the product line scope, core assets, and production plan have been established in the core asset base, and a few systems have been produced from that base, most of the work to support software system integration for the product line is complete. The interfaces have been defined, and they work predictably. They have been tested. Components work with one another. In subsequent variations and adaptations of the product, there is relatively little software system integration effort when the variations and adaptations occur within components. Even when new components are being added with new interfaces, the models from previous interfaces can and should be followed, thus minimizing the work and the risk of integration. So, in a very real sense, products (after the first one or two) tend to be “pre-integrated” such that there are few surprises when a system comes together.

### Application to Core Asset Development

When core assets are developed, acquired, or mined, remember to take integration into account during planning and budgeting. Evaluate any components you buy, mine, or commission for their integrability and granularity. A component is “integrable” if its interfaces (in the Parnas sense) are well-defined and well documented. Such a component may be integrated with other components directly through application programming interfaces (APIs) or potentially through wrapping. Finally, remember that it is generally easier to build a system from small numbers of large, pre-integrated pieces than from large numbers of small, unintegrated components.

### Application to Product Development

Software system integration is, or should be, the primary activity in product development within a product line. The core asset base consists of a relatively small number of large-grained assets covering requirements, architecture, components, test plans, test cases, and so forth, along with their respective attached processes. The core assets are engineered to work together in accordance with the product line architecture but still require tailoring and integration to build a product. The attached process guides tailoring and integration at the core asset level. The production plan provides guidance for developing a whole product.

A big benefit of product line practice is that integration during product development becomes a very predictable activity. In the product line, integration is based on the specific tailoring and integration guidance defined by the production plan. This generic production plan guides product developers in the specific steps they must take to tailor the full range of core assets needed in the production of their individual product. Included in this guidance is how to tailor the generic production plan itself for the individual product. Another benefit of product line practice is that software system integration costs tend to decrease for each of the subsequent products in the product line. If the production plan for a specific product calls for the addition of components or internal changes in components, some integration may be required depending on the nature of the changes. These changes are known up front and can be planned along with core asset integration. Finally, in the system generation case, integration becomes a matter of providing values for the parameters and launching the construction tool. The key in all these cases is that the integration occurs according to a preordained and tested scheme.

## Example Practices

**Interface languages:** Languages such as the Interface Description Language (IDL), Object Constraint Language (OCL), and Web Services Definition Language (WSDL) allow you to define interfaces in a way that can be automatically checked for consistency and completeness. Programming languages such as Java allow you to define a compilable specification separate from the body. Java programmers have found that keeping a continuously integrated system using full specifications and stubbed bodies decreases the integration time and costs dramatically. These languages and others do not allow the specification of the full semantic interfaces of components, but the integration bugs they allow you to catch early make using them pay off.

**Wrapping:** Wrapping, described as a specific practice in the “Mining Existing Assets” practice area, involves writing a small piece of software to mediate between the interface that a component user expects and the interface that the used component comes with. Wrapping is a technique for integrating components whose interfaces you do not control, such as components that you mined or acquired from a third party [Seacord 2001a].

**Middleware:** An especially integrable kind of architecture employs a specific class of software products to be the intermediaries between user interfaces on the one hand and the data generators and repositories on the other. Such software is called *middleware* and is used in connection with Distributed Object Technology (DOT) [Wallnau 1997a]. There are several prominent examples of middleware standards and technology such as .Net—Microsoft’s middleware to support distributed Web applications. Another collection of proprietary middleware solutions includes those that have grown around the Java programming language, such as Java 2 Enterprise Edition (J2EE). Software system integration involving Web services and service-oriented architectures make up another distributed object computing environment that product line integration must deal with. Middleware is discussed in more detail in the “Architecture Definition” practice area.

**System generation:** In cases in which all (or most) of the product line variability is known in advance, a new product in a product line can be produced with no software system integration at all. In these cases, it may be possible to have a template system from which a computer program produces the new products in the product line simply by specifying variabilities as actual parameters. Such a program is called a “system generator.” One example of such a family of products would be an operating system in which all the variabilities of the system are known ahead of time. Then, to generate the operating system, the “sysgen” program is simply provided with a list of system parameters (such as processor, disk, peripheral types, and their performance characteristics), and the program produces a tailored operating system rather than integrating all the components of an operating system.

**FAST generators:** Weiss and Lai describe a process for building families of systems using generator technology [Weiss 1999a]. The Family-Oriented Abstraction, Specification, and Translation (FAST) process begins by explicitly identifying specific commonalities and variabilities among potential family members and then designing a small special-purpose language to express both. The language is used as the basis for building a generator. Turning out a new family member (product) is then simply a matter of describing the product in the language and “compiling” that description to produce the product.

## Practice Risks

The major risks associated with software system integration include

- **natural-language interface documentation:** Relying too heavily on natural language for system interface documentation and not relying heavily enough on the automated checking of system interfaces will lead to integration errors. Natural-language interfaces are imprecise, incomplete, and error prone. As in single-system development, undetected interface errors increase the overall cost of integration. Errors in core assets that remain undetected until integration time also lead to significant repair costs, especially since an asset may be used in multiple systems. Automated tools, however, are more oriented to syntactic checking and less effective at checking race conditions, semantic mismatch, fidelity mismatch, and so forth. Some interface specifications must still be done largely with natural language and are still error prone.
- **component granularity:** There is a risk in trying to integrate components that are too small. The cost of integration is directly proportional to the number and size of the interfaces. If the components are small, the number of interfaces increases proportionally, if not geometrically, depending on the connections they have to each other. This leads to greatly increased testing time. One of the lessons of the CelsiusTech case study was that “CelsiusTech found it economically infeasible to integrate large systems at the Ada-unit level” [Brownsword 1996a]. Although the component granularity is dictated by the architecture, we capture the risk here, because this is where the consequence will make itself known.
- **variation support:** There is a risk in trying to make variations and adaptations that are too large or too different from existing components. When new components or subsystems are added, they must be integrated. Variations and adaptations within components are relatively inexpensive as

far as system integration is concerned, but new components may cause architectural changes that structure the product in ways that cause integration problems.

### Further Reading

[Alhir 2002a]
Alhir provides an overview of the Unified Process and its relationship to UML.

[Microsoft 2007a] & [Sun 2007a]
These Web sites provide a wealth of material on .NET and J2EE, respectively.

[Wallnau 1997a]
Wallnau, Weiderman, and Northrop provide a nicely digestible overview of middleware.

[Weiss 1999a]
Weiss and Lai describe the FAST process, which includes a generator-building step that essentially obviates the integration phase of product development.

## Testing

Testing has two main functions: (1) helping to identify faults that lead to failures so they can be repaired and (2) determining whether the software under test can perform as specified by its requirements. In certain domains and styles of development, testing has been performed to estimate the reliability of software.

Since it is almost always impractical to exercise a program against all possible inputs, testing is really a search process. During development, the developer tests by using those inputs that are most likely to result in failures. After the software under test has reached some stage of completion, the system tester searches for those failures the user is most likely to encounter. Not all failures have the same impact on the user. The amount of effort that is expended in searching for these failures should be proportionate to the impact on the quality of the program.

Testing is a continuous activity that cuts across all phases of the software development process. It is also a labor-intensive activity: Estimates of the resources expended writing code for testing purposes range from 40% to as high as 300% or even 500% of the amount expended for all other effort on the application under development [Pressman 1998a, p. 595]. This high cost makes testing an attractive target for improvement.

Different types of testing, such as unit, integration, and system testing, are carried out during the development process. Regardless of the type of testing, each task involved is organized around three basic activities:

1. **analysis:** The material to be tested is examined using specific strategies to identify appropriate test cases. Analysis techniques that involve structured artifacts such as architecture description languages and programming languages can be automated to reduce the test resources needed for a project. Performing test analysis will actually detect some defects such as poor testability. The output from this activity is a detailed test plan.

2. **construction:** The artifacts needed to execute the tests specified in the test plan are built. These artifacts usually include test drivers, test data sets, and the software that implements the actual tests. Commonalities among products and product parts support the development of frameworks and of harnesses that simplify test construction.
3. **execution and evaluation:** The tests are conducted, and the results are analyzed. The software is judged to have passed or failed each test. This information guides decisions about what the next step will be in the development process. The reuse of test cases across pieces of software amortizes the often high cost of test oracles—those software pieces that determine whether a test has passed or failed.

All testing activities should be carried out under the following desiderata:

1. **Testing is objective.** The process by which criteria are determined should be guided only by the satisfaction of the asset's requirements.
2. **Testing is systematic.** Test criteria are selected according to an algorithm that prescribes a reason for selecting each criterion.
3. **Testing is thorough.** The criteria used should achieve some logical closure that can be viewed as complete by some definition such as touching every line of code or executing every decision point.
4. **Testing is integral to the development process.** Before the software under test is produced, plans should be made as to how best to assess it.

Testing activities produce four major types of artifacts:

1. **test cases:** Selecting test cases is the fundamental test activity. Test cases are designed by setting a goal, achieving a certain level of test coverage, and then analyzing the item being tested to determine how to achieve that coverage. The test approach (e.g., find the highest risk or most likely defects) will direct the tester as test cases are selected. For each test case, the test context, input, and expected result are captured in a test plan, test data sets, and ultimately test software.
2. **test documents:** Test plans and test reports are the two primary types of test documents.
3. **test data sets:** The data needed for a test include all the inputs required to establish the preconditions for a test case and the actual test step. The construction and verification of these data sets require a significant resource investment.
4. **test software:** Test harnesses can be as complicated as the production software. For example, timing a component's response may be necessary to determine whether that component has met its real-time requirements. Or, it may be necessary to populate a large database, execute a test case, and then restore the database to its initial state for the next test case.

The remainder of this overview summarizes different kinds of testing. The first four types serve as exit gates for project phases.

**Design model validation:** Each phase in the development process that creates a model of the product or some portion of it should include testing activities that verify the syntax of the model and validate it against the required system. The test can serve as the exit criteria for that phase. We use "model" in a broad sense, to refer to non-software assets that represent a product for the purpose of either making predictions about the product implementation or prescribing constraints for other assets. A business case for a product line is a model; it predicts how profitable the product line will be. Software designs are models; they predict behavior and also impose constraints on implementations. Preeminent among the models is the software architecture, and its validation is so important that "Architecture Evaluation" is its own practice area.

**Unit testing:** Testing for implementation defects begins with the most basic unit of code development. This unit may be a function, class, or component. This kind of testing occurs during coding; therefore, the intention is to direct the testing search to those portions of the code that are most likely to contain faults—complex control structures, for example. As each unit is constructed, it is tested to ensure that it (1) does everything that its specification claims and (2) does not do anything it should not. A test case associates a set of input values with the result that should be produced by a correctly functioning system. The *functional testing* strategy uses the specification of the unit to determine which inputs to use in the testing. This strategy provides evidence that the unit does everything it is supposed to. A second strategy, termed *structural testing*, selects test inputs on the basis of the structure of the code that implements the functionality of the unit. This strategy provides evidence that the unit does not do anything it is not supposed to.

**Subsystem integration testing:** The integration of basic units, even those that have been adequately unit tested, may produce failures resulting from the interaction of the units. Timing discrepancies and type/subtype relationships can be the source of these errors. The tests are constructed from the use cases used to represent the full product's requirements. The integration test plan should describe tests that have been systematically selected from the interactions among the units being integrated. Protocol descriptions between pairs of units or flows through sets of units that implement a specific pattern of behavior can be used to select the test cases. Test cases should include instances in which the error-handling capability of the units is evaluated, such as when one unit throws an exception that should be caught by another unit.

**System integration testing:** When some critical mass of subsystems has been fully developed and tested, the focus shifts to representative tests of the completed application as a whole to determine whether a product does what it is supposed to do. These representative tests are selected to cover the complete specification for the portion of functionality that has been produced. The amount of testing a specific function receives is based either on its frequency of use (operational profiles) or on the criticality of the function (risk-based testing). Special forms of system testing include *load testing* (to determine if the software can handle the anticipated amount of work), *stress testing* (to determine if the software can handle an unanticipated amount of work), and *performance testing* (to determine if the software can handle the anticipated amount of work in the required time).

In addition to testing as the exit criteria for process phases, the next five types of tests described are applied to verify certain product properties.

**Regression testing:** Regression testing is used to ascertain that the software under test that exhibited the expected behavior prior to a change continues to exhibit that behavior after the change. Regression tests are constructed, and periodically applied, to determine whether the software under test remains correct and consistent over time. Regression testing is triggered by changes that affect a predefined scope of assets or that affect certain critical assets. The actual test cases used in regression testing are no different from any other test cases. The regression test suite is a sample of the functional tests from the original test suites administered prior to any changes.

**Conformance testing:** Conformance testing determines whether the software under test can be used in a specific role in an application. The conformance test set should cover all the required interactions between all the components that will participate in the application.

**Acceptance testing:** To validate the claims of the manufacturer or provider, the consumer performs acceptance testing. The acceptance test is more realistic than the system test, since the application being tested is sited in the consumer's actual environment.

**Deployment testing:** Deployment testing is conducted by the development organization prior to releasing the software to customers for acceptance testing. Where acceptance testing focuses on the functionality of the delivered product, deployment testing covers all the unique system configurations on which the product is to be deployed. This testing focuses on the interaction between the product and platform-specific libraries, device drivers, and operating systems. During the deployment testing phase, the application's ability to deploy or install itself is also tested.

**Reliability models:** Testing is used to estimate the reliability of a software component or system [Musa 1999a]; however, establishing the reliability of a piece of software through testing is a costly process. The test cases are selected based on the expected frequency of use of each product feature.

## Aspects Peculiar to Product Lines

Testing in a product line organization examines the core asset software, the product-specific software, the interactions between them, and ultimately the completed products. Responsibilities for testing in a product line organization may be distributed differently than in a single-system development effort [Clements 2002c, p. 130].

Also, unlike single-system development projects, testing is an activity (and producer of artifacts) whose output is reused across a multitude of products. Planning is necessary to take full advantage of the benefits that reuse brings. The following guidelines should help.

**Structure the set of testing processes to test each artifact as early as possible:** While the product line architects and implementers can focus on one variation point at a time, much of the testing work cuts across multiple variation points resulting in a potential combinatorial explosion of test cases. This potential can be mitigated by testing every artifact in the product line as early as possible in as isolated a context as possible. Doing so reduces the range of defects that must be searched for at each test point, thus greatly reducing the possible combinations.

**Structure test artifacts to accommodate the product line variation:** The test artifacts—test cases, test plans, test harnesses—should be as variable as the software that implements the product. The key to this variability is designing in the necessary variation so that the test artifact can be made to cover the complete range of product variability. Research evidence supports using the same variation mechanism used in the product implementation to implement variation in the test software.

**Maintain the test artifacts:** Structuring the test software to be used in multiple products reduces the cost of maintaining the test software, since it is easier to identify where to make changes. The development environment already contains tools that work with the application's units and can just as easily be applied to the units of test software. In iterative, incremental development, and in fact any development process that corrects its mistakes, the test code will be executed many times over the span of development.

**Structure the testing software for traceability:** The structure of the test software should support traceability from the test code itself to the code being tested. As changes are made to the product code, corresponding changes may be required to the test code. So, to maximize traceability, the test software should reflect the product line architecture where possible. Grouping the test code for a software unit from the application in a single unit of test software creates the mapping from test code to source code. For example, in an object-oriented development effort, the test software for a class is grouped within a single class. Where two parts of the product line architecture have a particular relationship, the test code for each of those parts is also related. For example, where one class in the application software inherits from another, the test class for that class inherits from the test class for the parent application class. In object-oriented software, the inheritance relation defines a hierarchy of definitions from abstract to specific in which each subclass adds more specific information to what it inherits. Test plans should be established at each of these levels, and test cases should be designed to have increasing specificity at each level of the hierarchy. The abstract test cases are not applied directly any more than abstract class definitions are used directly; however, they provide support for the reuse of definitions.

**Reuse product line assets for system integration testing:** The "Requirements Engineering" and "Architecture Definition" practice areas discuss various approaches to producing use cases and scenarios that describe how the system is intended to work. You should select test cases and inputs for system integration testing based on these descriptions.

**Automate regression testing:** Sampling original test sets to create regression test suites and their execution should be automated to encourage frequent retesting. The sampling algorithm should be weighted to test all variation points more than regions of commonality. The variation points are where the most changes, and probably the most errors, will occur.

**Expand the testing portfolio:** In a product line environment, testing is more cost-effective for the development organization, so more types of tests can be run. Stress, performance, and other types of narrowly focused tests can be added cost-effectively to the intended test suite, thereby reducing the possibility of failures in the field.

### Application to Core Asset Development

Testing concepts apply to core asset development in two ways. First, testing itself produces core assets. The four categories of artifacts (test cases, test documents, test data sets, and test software) mentioned earlier are core assets of the product line. They will be used for testing multiple products and pieces of those products.

The second way in which testing concepts are applied to core asset development is testing other core assets; this is a key activity in satisfying the quality and reuse goals in a product line effort. Building a component to be reusable is widely recognized to be more costly than a one-off implementation targeted at a specific application, because the component must be designed to handle a wide range of inputs and encompass a more complete set of states. (This applies to non-software assets as well.) This results in additional tests being necessary to achieve adequate test coverage; however, the scale of reuse in a product line effort keeps this testing cost-effective.

**Testing non-software core assets:** Every asset should be validated as it is created. This includes the business case, scope definition, and requirements model and carries on to the analysis, architecture, and detailed design models. To make the validation of these models more effective, the activity can be structured as a testing activity that embodies the criteria of being objective, systematic, thorough, and integrated. To ensure objective testing, those who created the model should not be testers.

For models built using a nonexecutable notation, the process described above is a review process. First, take full advantage of syntax checkers and other tools that provide some automation to validate certain aspects of the model. Then use the scenario-driven approach in which reviewers manually (with the help of those who created the model) trace through the model to determine the answer that would result if the model could be executed. An alternative is to actually build an executable version of the model.

**Testing software core assets:** Software core assets include components, or even complete applications, that are intended to be integrated to produce products. Subject each component to a rigorous test during its construction. Examine the component against its specification, but also examine the behavior of the component in integrated situations. Other software core assets are development tools. If they are built or significantly modified by the product line organization, they must be tested. Product line core assets have variation points, implemented perhaps by providing a parameterization mechanism or multiple implementations of an interface. The test plan for an individual component is divided into functional and structural test suites. Functional tests can be used for all the variations. The structural tests must be modified for each different variation as shown in Figure 7. For example, in object-oriented systems, the multiple implementations are related to some abstract definition via an inheritance relationship.

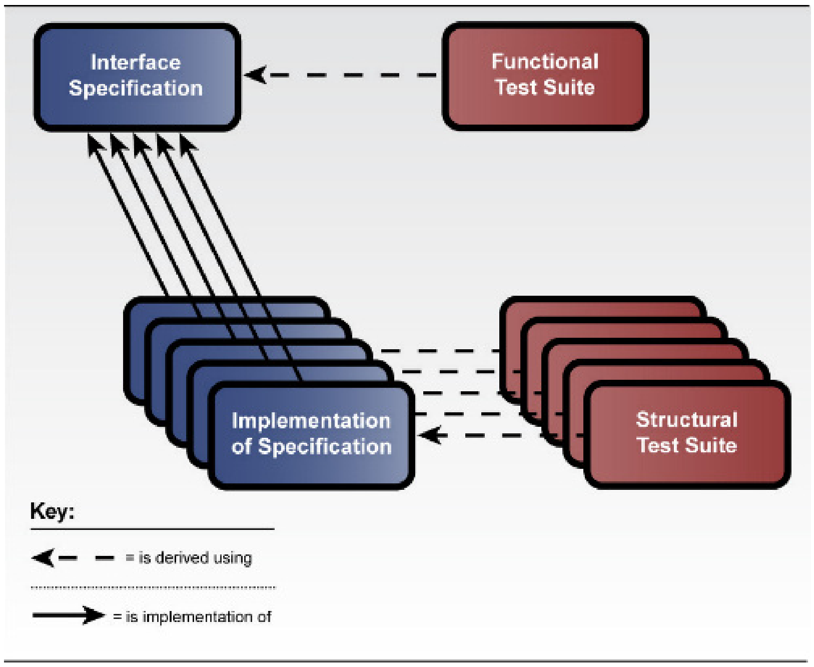


*Figure 7: Multiple Implementations Lead to Multiple Test Suites*

An acceptance test is performed on all the assets being obtained from outside the organization. Designing the acceptance test includes identifying the desired attributes, defining acceptable levels of those attributes, and evaluating the asset to see if it possesses those attributes. Assets such as compilers, other modeling tools from which code will be generated, and component libraries should be tested for accuracy and compatibility before being deployed to the technical staff.

## Application to Product Development

Testing is used in two fundamental ways for products: (1) between phases of the development process to *verify* that what was produced in the last phase is correct and suitable as input to the next phase and (2) to *validate* a product against its requirements. Validation tests are intended to evaluate correctness relative to requirements.

In a product line effort, the main product development activity is assembling products from core assets. The majority of each product's specification will be defined in documentation generic to the product line. Define a complete set of functional tests for that specification. Some portion of each product's implementation will also be created at the product line level. Even if the functionality has been tested via some mechanism at the product line level, when it is integrated into a specific product, interactions with product-specific functionality can lead to failures. Define a set of interaction tests that will ensure that the additions made by the product developers do not cause failures. The tests used at the product level can be derived from the functionality tests, or possibly test templates, created at the product line level. The mapping between product core assets and testing assets facilitates the reuse of these test assets. The mapping associates the test cases, as well as the test drivers and test data sets, with those requirements that are common across the product line. By taking advantage of this commonality, the amount of effort associated with testing and retesting a product is reduced.

There is a tradeoff between saving resources through the reuse of testing assets and improving quality by expending some of the saved resources on additional testing. By devoting more resources to those products created early in the product line effort, the quality of all deliverables is improved. Since these are the assets that will be reused, this improved quality will be propagated to future versions of the product being constructed and to other products in the product line.

## Example Practices

**Architecture evaluation:** Evaluating the product line architecture is a kind of testing, under the broad sense of the term used in this practice area. Architecture evaluation is covered in its own practice area.

**Build support for testing into components:** The product line architecture can provide support for testing by levying requirements on the systems' components. This support takes forms such as special test interfaces that allow self-test functionality to be invoked and special access to certain state types that are stored (maintained internally) by the program. These types of interfaces and functionality are often too resource intensive to be provided in a one-off system but are cost-effective in a product line environment.

The self-test functionality provides the system user the capability to determine whether the system is currently operational. This capability is particularly useful in systems that are configurable, systems that dynamically incorporate resources into the product, and systems that have a significant hardware component. The basic support for self-testing can be defined in the product line architecture and then elaborated by specific products. The self-test functionality can run a set of regression test cases that are designed to exercise those parts of the program that dynamically load and link functionality and those parts that rely on information from configuration files and other external resources.

**Guided inspection:** For analysis and model reviews, guided inspection is a technique that combines the checklist of an inspection with the thoroughness of testing [McGregor 1999a]. The inspection process is "guided" by the test cases. Other methods for guided inspections are discussed in the "Architecture Evaluation" practice area.

**Test-Driven Development (TDD):** TDD is one practice of the agile development community. Its goal is "clean code that works" [Beck 2002a]. In this practice, developers define requirements for the piece they are assigned to construct by maintaining close communication with the developers who are constructing related units and writing test cases that serve as the specification for the unit. The developer then writes and revises code until the unit passes all the tests. The rhythm of TDD is very short cycles of these steps:

- Define a new test.
- Execute all tests.
- Write code to fix tests that fail.
- Execute all tests.

TDD has the advantage that the test code is always in synch with the product code, because the test code defines the product code. The disadvantage of TDD is that there is not a good method for determining whether the set of test cases is complete, since the completeness of a test set is usually determined by comparing it to the specification.

TDD is applicable to product line organizations provided it is applied to units that are first defined in the context of the product line architecture. TDD does not provide tools and techniques for balancing the diverse quality attributes usually present in a product line. TDD can be successful if applied to units that a small group of developers, often a two-person or pair programming team, can produce in a timely manner. The range of variability for the unit should also be sufficiently narrow to allow for timely completion. The success of TDD depends on the availability of tools, such as JUnit, to assist with development and automate testing.

### Practice Risks

The major risk in testing is not doing enough of it and not doing it in high-payoff ways. Inadequate testing will result in low software quality, which will undermine the success of the product line. Inadequate validation of non-software artifacts will result in a loss of trust in those artifacts and a decaying of the process-based or documentation-based practices they were intended to support. Specific testing risks include

- **inadequate unit testing:** Component quality will be low if the unit-level testing is inadequate. Often technical staff will decide to "save time" by performing little or no unit-level testing. This may actually take more time, because it will require an unexpected amount of integration and system testing. Further time may be lost, because it is well established that repairing errors found late in the development cycle is more costly than repairing those found early. The probability that this risk will occur is lower in a product line environment that fosters a culture of reuse. However, if the risk becomes a problem, the cost will be far greater. The increased cost is due directly to the propagation of poor-quality components across the larger number of reuse sites. A well-defined software development process that specifies a unit test activity and defines a level of adequate coverage mitigates this risk.
- **inadequate unit testing due to inadequate tool support:** The automated unit testing will be inadequate if it is conducted on application program interfaces (APIs) only. Few of the automated testing tools work on APIs, and those that do usually require some amount of custom programming or comprehensive specifications. The risk is that, if adequate tools are not available, more resources will be needed to achieve an acceptable level of coverage. The probability that this risk will occur is less in a product line environment where the cost of building special tools can be amortized over multiple products. However, if the risk becomes a problem, the cost will be far greater due to the propagation of poor-quality components across the larger number of reuse sites. A tools group at the product line level that provides testing support to all products mitigates this risk.
- **inadequate specifications:** The testability of components will be low if inadequate specifications make it impossible to design tests. The probability that this risk will occur is about the same in a product line environment as it is in a single-system one. However, if the risk becomes a problem, the cost will be far greater in a product line environment—a cost that includes increased

resource requirements for ensuring adequate quality. Training developers to write complete, consistent, and correct specifications mitigates this risk.

- **insufficient integration testing:** The flow of products will be slower than expected if sufficient integration testing is not conducted. The probability that this risk will occur will be higher in a product line environment if the product line team and the product teams are not linked in a feedback loop. Internally, the product line team should use the product line architecture as a blueprint for communication links between component development teams to ensure that the interactions between components will be complete and correct.
- **testing too late:** The leverage gained from testing a product line is at its peak when applied early. The later the tests are applied, the more combinations there are to test. By testing early and applying incremental integration tests, far fewer tests will be needed.
- **testing too much:** The test plan must have clear stopping criteria. Experience will yield the usual defect densities (number of defects per line of code). When testing and repair have achieved that level of defect density, further effort may not have a positive return on investment (ROI). In the absence of experience, establish test coverage levels and stop when they are achieved.
- **inadequate test infrastructure:** The anticipated high level of test asset reuse will not be realized unless sufficient resources are devoted to the test infrastructure. If developers are allowed to test in ad hoc ways or the test software architecture is not maintained properly, new tests will not be derived from existing ones. The resulting loss may be a reduction in quality and available resources.

## Further Reading

[Beck 2002a]
Beck's book introduces TDD.

[Beizer 1990a]
Beizer provides a comprehensive survey of testing techniques applied at the unit, integration, and system levels. He describes basic techniques regardless of the process or development paradigm. His book serves as good general background.

[McGregor 2001a]
McGregor provides a jump-start for personnel charged with establishing the testing process for a product line environment. He presents techniques for taking advantage of the personnel organization and software architectures in order to reduce the effort required for adequate testing. The techniques organize unit-level testing assets in a manner that directly reflects the architecture of the product software. The techniques also associate the requirements, in the form of use cases, with the system test cases. The proceedings of the Software Product Line Testing (SPLiT) workshops also provide insight into the research directions in this area [SPLiT 2004a, SPLiT 2005a].

[Musa 1999a]
Musa ties the amount of testing to measures of reliability. He describes the computation required to determine the levels of tests that are required to "prove" specific levels of reliability.

[SPLiT 2004a] & [SPLiT 2005a]
These proceedings of the SPLiT workshops describe current research in testing practices designed specifically for product lines.

## Understanding Relevant Domains

One of the constants we've observed in successful software product line organizations is that they have at their disposal extensive experience in the domains that are relevant to their software endeavors. This practice area is about achieving that understanding. Domains are areas of expertise that can be applied to the creation of a system or set of systems. Domain knowledge is characterized by a set of concepts and terminology understood by practitioners in that area of expertise. It also includes an understanding of recurring problems and known solutions within the domain. Knowledge from several domains is usually required to build a single product. For example, to build a distributed banking application, you would need knowledge of banking practices, commercial bank information systems, workflow management, database management systems, networking, and user interfaces, just to name a few. You would never attempt to build a distributed banking application (or any other nontrivial system) without first making sure you knew enough about the relevant business and technical areas to impart the expectation for success. "Knowing enough" to make good product decisions comes from understanding the relevant domains.

The practice of understanding the relevant domains involves

- identifying the areas of expertise—domains—that are useful for building the product or products under consideration
- identifying the recurring problems and known solutions within these domains
- capturing and representing this information in ways that allow it to be communicated to the stakeholders and used and reused across the entire effort

How does an organization achieve this understanding? Typically, it builds up its store of expertise with its prior experiences in delivering products. An organization can also hire outside experts who provide or augment the organic level of understanding. In addition, it can employ domain analysis methods to gather, organize, and communicate domain information in a form known as a domain model.

Understanding relevant domains is initiated by eliciting domain information from various sources. This elicitation includes the investigation of current products and interviews with domain experts and product marketers, developers, and users to identify the current and potential future capabilities for the product(s) being considered. The elicitation also captures the needs and expectations of the various stakeholders and helps in the assessment of the technical maturity and stability of the relevant domains. The extent of the elicitation activity depends on the availability of sources and the amount of domain knowledge already known in-house.

Domain understanding should include whatever key domain information can be used as the vehicle for analysis and reasoning. Domain information should capture different views of the product(s) from the perspectives of relevant stakeholders. In the absence of this information, your organization is vulnerable if experts leave. Further, documentation will refine and sharpen the understanding of the experts themselves; writing a thing down requires a deeper understanding of it than carrying it around in your head. And finally, recorded domain information can be reviewed and improved by stakeholders who can speak to their future needs.

The extent to which a formal domain analysis is performed to achieve domain understanding depends on these two factors:

1. **the depth of the organization's domain experience:** Organizations that have deep domain expertise frequently opt not to invest in a full-blown "formal" domain analysis to model their understanding of the relevant domains. (For example, they may use a hybrid approach that combines requirements gathering with use case modeling and commonality and variability analysis.) Foregoing formal domain analysis allows them to move into design more quickly or to perform some initial design activities (e.g., the investigation of architectural patterns) in parallel with the analysis.
2. **the amount of resources that can be devoted to analysis:** The time, money, and people allocated to the analysis will determine the duration and scope of the analysis activities. However that's not to suggest that more analysis is always better. In fact, "analysis paralysis" is a risk associated with this practice area.

Regardless of its formality or comprehensiveness, domain information should be documented to capture the expertise needed to serve as authoritative criteria for making product and design decisions. You need to have enough information to make sound business decisions without necessarily undertaking an extensive analysis of all the applicable domain knowledge. An organization has achieved a sufficient understanding of the domains relevant to a product or products when it can successfully apply that understanding towards

- reasoning about the technical and business implications of a proposed design
- making informed decisions about which features, capabilities, and technologies to offer in proposed products
- creating a set of artifacts that exploits the understanding of the relevant domains

## Aspects Peculiar to Product Lines

Understanding the relevant domains is the first step to defining the commonality and variability that can be expected to occur among the products identified in the product line's scope. Domain understanding for a product line emphasizes the commonality and variations present among the products, whereas domain analysis for a single system focuses instead on the technical concepts inherent in a single system or how that system might evolve once fielded.

Domain information for a product line will help you determine

- which capabilities tend to be common across systems in the domain(s) and what variations are present. This information will inform the process of scoping, in which the commonality and variations for *your* product line will be established.

- which subsets of capabilities might be packaged together as assets for the product line. This information begins to inform the architecture creation effort for the product line, by suggesting potential subsystems that have occurred in other systems in the domain(s).
- what constraints (e.g., standards, legal restrictions, business constraints, specific hardware platforms) apply to systems in the domain(s)
- which assets typically constitute members of the domain(s). This suggests a list of assets that an organization could begin to search for in its own legacy inventory or on the open market.
- whether to continue with the product line development effort

The last item is particularly important; the ability to reason about a product line can help management gain confidence in the soundness of the decision to adopt a product line approach. Put another way, domain understanding informs both the product line's scope and its associated business case. Domain understanding also grounds the design decisions in experience (even if the experience is not the organization's own), which reduces technical risk and also has a soothing effect on nervous managers. Even a rudimentary and informal domain analysis can help refine both the scope of the product line and the initial estimates for resource allocation. And it can show the organization and its management that they are not exploring uncharted territory.

The level of detail and degree of formality of the documented domain information for a product line depend both on the depth and distribution of an organization's domain knowledge and on the kind of reasoning about the product line that the organization wishes to support by modeling. Some organizations have such a thorough understanding of their domains and their reuse potential that they can move very quickly from identifying a business opportunity to creating core assets. Organizations less experienced in exploiting reuse need to approach the problem more deliberately, analyzing commonality and variability to understand the technical implications for assets and products and the business implications for the product line as a whole. Additionally, they may have to hire domain experts if the requisite domain expertise is missing or immature. Whatever the degree of understanding, each organization has some model of the product line that evolves as the development effort progresses. It may be an informal shared understanding of the product line with minimal documentation, or it may be a more formal domain model—that is, an abstract characterization of the product line, complete with its own representation schemes and automated support. If the purpose of the analysis is to obtain insights into the technical feasibility and potential scope of the product line quickly, the analysis does not need to model the commonality and variability exhaustively.

In summary, understanding relevant domains is about systematically capturing and using knowledge about systems similar to the ones that you are about to build. This knowledge provides you with an informed world view from which you can then make specific decisions about your product line, especially regarding its scope, shared and unique requirements, and architecture.

### Application to Core Asset Development

An understanding of the relevant domains is applied to core asset development in order to identify and model the opportunities for large-grained reuse across a product line early in its life cycle. This knowledge informs the articulation of common product constraints, as described in Core Asset Development. It has a profound influence on the architecture for the product line, which is the definitive design statement about the commonalities and variabilities that will be supported across the product line. It also supports the business case for the product line and feeds the “Scoping” and “Requirements Engineering” practice areas.

### Application to Product Development

An understanding of the relevant domains supports the assessment of customer-specific requests for new product line capabilities. This assessment, in turn, may broaden your understanding of the relevant domain knowledge. In fact, such decision-making opportunities represent one form of feedback between core asset consumers and core asset producers, and they have the potential to expand the scope of the product line by challenging previously held assumptions about commonality and variability. Any such feedback from products to core assets should be documented so that it can be evaluated and, if necessary, incorporated into any models of the relevant domains.

Similarly, an understanding of the relevant domains can also be used as the basis for analyzing the effects of a proposed change in the product line’s requirements and for recommending a course of action. For example, a business decision to switch to a different operating system can be checked against the operating system capabilities assumed for the product line. At a minimum, there could be a checklist of features whose presence or absence in the new operating system will either shorten or lengthen the product development schedule. A more detailed model of assumed operating system capabilities would provide a greater ability to quantify the effects of the proposed change on the product development process.

### Example Practices

**Have domain experts available:** The best way to achieve domain understanding is by having the right people in the product line organization—those with extensive experience in the relevant domains. (This was a recurring theme at the one of the SEI product line practice workshops [Bass 1998a].)

**Scope, commonality, and variability (SCV) analysis:** The SCV approach, from Lucent Technologies, gives software engineers a systematic way of thinking about the product family they are creating. It identifies, formalizes, and documents commonalities and variabilities. SCV is the commonality analysis portion of the Family-Oriented Abstraction, Specification, and Translation (FAST) approach, also from Lucent Technologies. The FAST method includes the dual life cycles of domain engineering and application engineering. It uses the results of a commonality analysis to create a language for both specifying domain members and generating them from the specification [Coplien 1998a, Weiss 1999a].

**Domain analysis and design process (DADP):** DADP is a process model for domain analysis and design created by the Defense Information Systems Agency (DISA). It is based on an object-oriented approach to analysis and design. The domain analysis process focuses on identifying commonalities and determining common object adaptation requirements (the differences among domain common objects are not described in terms of variability, but rather in terms of tailoring the information to particular needs) [DISA 1993a].

**Feature-oriented domain analysis (FODA):** The FODA method defines the process and products of a domain analysis, with an emphasis on the commonality and variability of the features that users commonly expect in domain applications. The analysis process creates models of the domain that describe its relationship to other domains, its common and variant features, and the behavior of the applications in it [Kang 1990a, Cohen 1991a]. The feature-oriented reuse method (FORM) is an extension to FODA that addresses the design and implementation phases [Kang 1998a].

**Synthesis process of the reuse-driven software processes (RSP) approach:** Synthesis is a methodology for creating software systems as instances of a family of similar systems. It was developed by the Software Productivity Consortium in recognition of a need to produce improvements in software productivity, product quality, manageability, and customer responsiveness. A Synthesis process consists of two subprocesses: domain engineering and application engineering. The domain engineering process includes an explicit domain analysis process that captures commonality and variability for a product family [SPC 1993b].

**Domain analysis process of organizational domain modeling (ODM):** ODM is a highly tailorable and configurable domain engineering process model. It's organized hierarchically as a tree of processes, and while it doesn't use the term *domain analysis*, all the elements of a domain analysis process are present at various levels within its process tree. In particular, the model contains subtrees of processes for domain identification and scoping, domain modeling, and model refinement [STARS 1996a].

There are documented cases of organizations that have employed and adapted these methods successfully. Several books contain chapters that summarize and compare the characteristics of these and other domain analysis methods [Arango 1994a, Wartik 1992a]. There are also cases where organizations have employed domain analysis practices successfully in the creation of product lines, even though they did not use a documented domain analysis method [Bass 1998a].

The elicitation, representation, and validation of relevant domain information and the techniques employed (e.g., object-oriented technology, use case modeling, state-transition diagrams) vary across the different domain analysis methods and are not described using the same vocabulary. In a product line context, organizations perform these practices to varying degrees of completeness and rigor. In particular, organizations with deep domain knowledge and considerable expertise in applying it to developing products often opt for an abbreviated form of analysis that proceeds very quickly to design.

**Product Line Software Engineering Customizable Domain Analysis (PuLSE-CDA):** This method is one of the elements of the PuLSE product line engineering framework. PuLSE-CDA creates a domain model that can be customized to specific projects based on variation points called customization decisions. It also creates a decision model (based on that of Synthesis [SPC 1993b]) that provides instantiation support for specifying the products in a product line. There is a tool called Domain and Variant Engineering Supporting Technology/CDA (DIVERSITY/CDA) to support the method.

PuLSE was developed by the Fraunhofer Institute for Experimental Software Engineering (IESE) [Bayer 2000a].

**Domain-Specific Modeling (DSM):** DSM creates a language based on high-level domain concepts and supports the automatic generation of programs specified in that language. Tolvanen identifies over 20 industrial cases in which this approach has been successfully applied to target languages such as C, C++, Java 2 Platform, Enterprise Edition (J2EE), and Extensible Markup Language (XML) [Tolvanen 2005a]. The domains of these industries include telecom services, insurance, medical device configuration, handheld devices, machine control, and business processes.

## Practice Risks

Inadequate domain understanding in the organization will jeopardize the product line effort. Without a detailed (not just basic) understanding of the precise commonalities and variabilities that need to be accounted for, the architect's hands are tied, the business case will be weak, and the scope definition will be unrealistic. The result will be a set of products that do not adequately address the needs and market demands they were intended to meet. Inadequate domain understanding can result from

- **analysis paralysis:** The "analysis paralysis" phenomenon occurs when an inordinate amount of time is devoted to the creation of one or more very detailed analysis models. A strategy to mitigate this risk is to perform a relatively quick, broad exploration of commonality and variability to gain an understanding of the issues and their effect on the product line. This allows for early input to management decision makers, who can then assess the value of proceeding further with the analysis and allocate resources accordingly. It also allows some of the design work to proceed in parallel with, for example, the initial architecture exploration. Ideally, this approach would be part of an overall iterative and incremental process for the development of the product line. An alternative risk mitigation strategy is to narrow the scope of the analysis.
- **a lack of access to the necessary domain expertise:** Eliciting domain information from the domain experts is essential. However, they are usually coveted organizational resources who are in high demand and may not be in the same geographical location as the rest of the analysis team. Therefore, careful planning is needed to ensure that the domain experts' time is not wasted. Such planning may include scheduling specific meetings with the experts, circulating elicitation questionnaires in advance, and using videoconferences or teleconferences when face-to-face meetings are not possible. In all cases, there must be management commitment to ensure that the domain experts are available for participation.
- **inadequate documentation and sharing of relevant domain information:** If the understanding of the relevant domains is in the heads of a few key people and not shared with the rest of the product line team, there is a great and obvious risk should any of these key people leave the organization. There is also the more subtle risk of false assumptions and time wasted rediscovering what is already known. Hence, the "mental model" carried by the key people should at least be recorded so that it can be shared. The level of rigor and amount of detail in the documentation should be driven by the need to make such information robust enough for the long haul of a product line effort. At a minimum, assumptions and decisions about what is common, what is variable, and what is excluded from the product line should be documented, plus some justification of

them that ties to the business case. Recording this information will also mitigate the risk of having key people leave the project, taking their domain understanding with them.

- **a lack of an understanding of an analysis process:** The naïve application of an analysis process can be evidenced by a too-early focus on design and implementation issues or too much time spent on the analysis. Proper training in both the analysis method and its role for the product line is essential here; the analysis should be performed in the context of the organization's overall reuse and process improvement goals and the specific goals established for the product line.
- **a lack of the appropriate tool support for the process and products of domain analysis:** Organizations typically use existing commercial object-oriented analysis and design tools to support a product line analysis effort. Such tools may not support the kind of conceptual modeling required by product line analysis; care must be taken to avoid being driven by the tool into a premature design process masquerading as an analysis process.
- **a lack of management commitment:** If management does not appreciate the value of domain understanding and does not understand the need for an analysis process that could be time-consuming and that does not culminate with the production of any marketable products (or executable code), the effort will likely lose requisite management support. The documentation of relevant domain information and any analysis that is initiated should be grounded in issues that are key to management's ability to support specific business goals established for the product line. The product line organization must make upper management aware of the importance of having documented domain understanding not only as essential input to subsequent decisions about the product line but also as a powerful way of mitigating the technical and product risks associated with the product line effort.

### Further Reading

[Arango 1994a, Ch. 2]
Arango provides a comparative survey of published domain analysis methods that also maps each method onto a common domain analysis process.

[Ardis 2000a]
Ardis and colleagues describe an effort in understanding relevant domains at Lucent Technologies in 1994.

[Bayer 2000a]
Bayer and colleagues describe CDA, a domain analysis method that is part of the PuLSE product line engineering framework created by the Fraunhofer Institute for Experimental Software Engineering.

[Geppert 2003a]
Geppert and Weiss describe Avaya Labs' method for the quantitative assessment of potential product line domains and, using an example application of this method, define measures for a candidate domain's potential corporate impact and likelihood of success when implemented as a product line.

[Kang 1990a]
Kang and colleagues provide a comprehensive description of FODA.

[Kang 1998a]
This paper presents FORM, an extension of FODA used to develop domain architectures and components for reuse.

[Lee 2000a]
Lee and colleagues describe their experience with domain analysis for their software product line.

[SPC 1993b]
This guidebook describes the Software Productivity Consortium's Synthesis methodology, which incorporates domain analysis into the construction of families of systems having similar descriptions.

[STARS 1996a]
This STARS report describes ODM.

[Tolvanen 2005a]
Tolvanen and Kelly describe industrial cases where domain-specific modeling has been used successfully.

[Weiss 1999a]
Weiss and Lai describe the commonality and variability analysis, which is an important part of Weiss and Lai's FAST process for product lines.

## Using Externally Available Software

The software for any system may be obtained in ways other than by developing it from scratch. Acquiring commercial, off-the-shelf (COTS) software, open source software, freeware (such as that from the Free Software Foundation), and Web-based services to populate service-oriented architectures are all promising options for organizations seeking to use already-existing software. Organizations do this in an effort to improve the cost-effectiveness and efficiency of their software development efforts [SEI 2007c, OSI 2007a, FSF 2007a].

Government software acquisitions also use the general terms *commercial item* (CI) and *non-developmental item* (NDI) to distinguish items, including software, that were developed by an organization other than the acquiring organization [FAR 2005a]. The term NDI is actually subsumed by the term CI.

The open source approach is the major alternative to acquiring proprietary software [OSI 2007a]. This approach is often more flexible than COTS but has distinct costs. Open source software is often accompanied by a license (discussed below) and sometimes comes with a formidable learning curve. Open source represents more than just a means of obtaining software: it is an important ingredient in the business model for a significant portion of the software market.

Many of the same opportunities and challenges apply to all the externally available software options (hereafter known as EAS). Other practice areas touch on EAS issues as well. Licensing issues are addressed in the "Developing an Acquisition Strategy" practice area. Integrating EAS with other software is a major challenge, as is testing it; see the "Software System Integration" and "Testing" practice areas, respectively. The “Mining Existing Assets” practice area covers the modification process of these items, once they’re onboard.

The use of EAS components has mushroomed in large-scale system development. Communication infrastructures, network management software, databases, and a host of domain-specific utilities are available both as COTS and open source components. Many of these utilities are packaged as services and used to populate service-oriented architectures. For example, suppose that you’re building an online auction Web site. You can find packages (or services) to manage the auctions, take credit cards for payment, and implement personalized search and notification functions for your customers. The list is endless. In many cases, entire architectures are now available in the marketplace. Using EAS is reported to cut costs, take advantage of common architectures, and enable large-scale reuse. However, when EAS is used in a system, you have far less control over how the components fit into the architecture and evolve. Using EAS introduces a new set of issues, concerns, and tradeoffs, many of which are listed in the “Practice Risks” section of this practice area.

Although specific practices vary, using EAS always necessitates the following steps:

1. **Analyze for architectural compatibility:** Determine the ways in which the software will fit into the architecture, paying particular attention to points of flexibility and variability, as well as areas that can’t be compromised. Conversely, recognize that the availability of EAS may represent an advantage of such magnitude (over in-house development) that it might be worthwhile to modify the architecture to make a place for that software.
2. **Understand the requirements of the organization:** Some organizations have specific technical constraints or standards to which all software must conform. These constraints may rule out otherwise acceptable EAS. On the positive side, such constraints will quickly narrow the search for suitable candidates. In any case, the constraints need to be written down so they can be reviewed for completeness, be revisited occasionally for relevancy or revision, and be given to someone to guide the search for EAS.
3. **Study the marketplace in detail:** New products are being developed all the time; be alert to those that can be relevant to your product line. For large-scale efforts, it will pay to dedicate resources to keeping up-to-date on the new technologies, middleware products, and relevant EAS that are available in this rapidly changing field. Try following relevant standards groups and products conforming to standards. Determine subsets of the marketplace that are important, subscribe to newsgroups and lists that advertise new components, and maintain a list or database of potentially relevant components for when it’s time to make a purchase decision. These activities are part of the “Technology Forecasting” practice area.
4. **Develop requirements in a flexible manner:** EAS is written to the vendor’s expectations about the market, not to your organization’s specifications. The vendors hope they meet your needs, but they probably won’t do so exactly. Instead of traditional requirements that specify “must” and “should” needs, requirements for EAS articulate broad categories of needs and possible tradeoffs, in addition to the critical or nonnegotiable requirements. These flexible requirements can be used to narrow the field of candidates; if none are found and the fallback is in-house development, the requirements can be tightened as usual to drive that development. But the bottom line is that you

need to keep an open mind about what you thought you needed, so you can take advantage of what's available.

5. **Develop an approach for the evaluation of products and technologies:** The approach should prescribe not only the evaluation steps but also the evaluation criteria. Include things such as price, vendor stability, the level of support required and provided, the ease of replacing the component with a competitor's product should the need arise, the simplicity of integration, and the ease of use. Be sure to include the important quality attributes that you expect the products to have: performance, security, reliability, and so forth.
6. **Select viable products and technologies:** Use the evaluation approach to qualify potential product candidates and make selections based on the evaluation criteria.
7. **Buy the products:** Buying software may not be simple. Organizational purchasing policies and guidelines must be followed. In some cases, you may want to test the products in prototypes before committing to a full purchase. In other cases, you can make a commitment immediately. Even when the EAS is free, following purchasing policies may prevent hasty decisions that are costly in terms of lost time and opportunities.
8. **Integrate products into the architecture:** Don't underestimate the task of integrating EAS with other EAS and with the rest of the system. Don't assume that EAS can simply be plugged into an existing architecture without modifications (of either the architecture or the product). Plan for adaptation with wrappers, middleware, or other software "glue." See the "Software System Integration" practice area for more advice on this topic.
9. **Test the products and their configuration:** The inclusion of EAS requires testing of the products' interfaces: remember that the components are essentially black boxes. Using EAS also requires testing any potential interactions with other components that may have an unpredictable impact. This is especially true for quality attribute requirements. Because EAS is developed by someone else according to a set of requirements unknown to you, quality attributes need to be considered specifically if factors such as security, performance, and availability are critical.
10. **Manage the system on an ongoing basis:** Monitor the current configuration of EAS, and scan the horizon for new products or potential replacements. Write guidelines for making decisions about upgrading and replacing components. Maintaining EAS means incorporating new releases into an existing set of core assets; therefore, maintenance requires attention to different upgrade cycles, potentially incompatible data sets, conflicting naming conventions, and new conflicts between different EAS. In addition, decide who will be the point of contact with the EAS vendors for issues, problems, and upgrades.

The discussion so far has related to using EAS in the form of components or services. It is also possible to adopt whole architectures and application frameworks off the shelf. A framework is a template for systems within a particular domain [Jacobson 1997a]. In the object-oriented world, a framework is often provided in the form of a set of class definitions. If a commercial or open source architecture is selected, it must be analyzed carefully to determine whether it is appropriate for the desired system. Perform an architecture evaluation just like you would for a homegrown architecture. The evaluation can likely be abbreviated, since many of the architecture's quality and behavioral attributes should already be known. An off-the-shelf architecture often admits the use of whole sets of available off-the-shelf components that are compatible with it.

Using an application-standard architecture brings about a subtle but significant change in focus. Instead of designing an architecture, specifying components, and then going to the marketplace to see what is available, evaluating the choices, and so on as described above, the marketplace is scoured for existing architectures and components and for systems built from both. The architecture is then selected or defined on the basis of the available EAS. From this perspective, EAS is used as product constraints that get factored into the requirements and early architectural activities and that greatly influence or even dictate the architecture [Albert 2002a].

## Aspects Peculiar to Product Lines

EAS can populate the core asset base of a product line, or it can be relevant for the development of specific products in the product line. If they are core assets, EAS, like other product line assets, must be flexible enough to satisfy the variability needs inherent in a product line. Variability, then, must be added to the selection criteria. When selecting the appropriate EAS, you need to look at the variety of products it will be used in. Pay particular attention to their variation points and any requirements that can't be compromised.

Because of the central importance of the architecture for product lines and because of the need to fit a potentially large family of systems, a product line solution using EAS needs to be generalized, involving general-purpose integration mechanisms that span a number of potential products. As a result, the range of potentially qualifying products may be reduced, and the use of wrappers and middleware must focus on generalized solutions rather than on some of the more opportunistic solutions that may be appropriate for single-product systems. The evolution strategy for making updates is now more complex, because the range of affected systems is greater.

## Application to Core Asset Development

EAS is certainly a viable choice to serve as core assets. Commercial components such as databases, graphical user interfaces (GUIs), graphics components, World Wide Web browsers, Java-based products, and other middleware products based on approaches such as .NET, remote method invocation (RMI), or their open source counterparts can represent an important part of the core asset infrastructure. Application- or domain-specific components or subsystems are also available. Because core assets need to be dependable over a long period of time, you'll need to pay attention to stability factors such as

- **maturity and robustness of the products:** Do they have a track record in the field, or will you be the first organization to use them?
- **expected product update schedule:** Will the vendor be issuing new releases every month? Every year? Every five years? Will the new versions be backward compatible, or will incorporating them require changes to other software in your system?

- **interoperability with a wide variety of products:** Do they work and play well with other assets, or do they make self-centered assumptions, such as the component that assumes it is the sole owner of the software's main timing loop?
- **stability of the vendor:** Is the vendor a trusted name in the field, or is it a new venture that is struggling to get off the ground and might be out of business in a year?

When evaluating EAS for adoption as core assets, consider the interfaces and the relevant protocols and standards to which the products conform. They will determine how well (or if) components will work within an infrastructure. Be aware that the mechanisms by which a product conforms to a standard may be idiosyncratic or inconsistent. In addition, since core assets can apply to a potentially large class of products, you should focus their testing on issues of generality and extensibility.

Licensing issues may be more complex when EAS is used as a core asset. Trivially, the license becomes a deployment item that may require user acceptance during the installation process. Ordinarily, a COTS license would be granted to use the product in multiple copies of a single system. But if the COTS product is a core asset, it will be used in multiple copies of many different systems, and the vendor is likely to want a different legal arrangement. There are many types of open source licenses. Even freeware may come with licensing restrictions. Services may require a fee-per-use that results in a prohibitively expensive proposition. The critical areas to consider are those where the free software interacts with proprietary software.

### Application to Product Development

An individual product in a product line may contain product-specific code that applies only to that product. If that code corresponds to a full component, there is no reason why that component cannot be EAS. Of course, the economics of such a decision will have to be weighed; in that case, the decision will be the same as for EAS use in a single-system development. Beyond that, the technical factors for choosing EAS are the same as for a core asset.

### Example Practices

COTS-based systems practices: The SEI developed a set of practices to help with COTS product and technology evaluation, COTS product acquisition and management, design and software engineering using COTS products, business case analysis, and COTS policy and planning [SEI 2007c].

Many of the guidelines apply not just to COTS but to other forms of EAS as well. For example, Carney suggests the following three-step approach for evaluating commercial products and technologies to see if they present a viable option for your development [Carney 1998b].

1. Plan the evaluation.
   - Define the problem, considering the entire context of the product line core assets (functional, technical, quality attributes, platform, resources, alternatives, and business issues).
   - Define the outcomes of the evaluation.
   - Assess the decision risk.
   - Identify the decision maker.

  - Identify the resources.
  - Identify the stakeholders.
  - Identify the alternatives.
  - Assess the nature of the evaluation context.
2. Design the evaluation instrument.
  - Specify the evaluation criteria.
  - Build a priority structure.
  - Define the assessment approach.
  - Select an aggregation technique.
  - Select assessment techniques.
3. Apply the evaluation instrument.
  - Obtain products (for evaluation).
  - Build a measurement infrastructure.
  - Aggregate data.
  - Form recommendations.

Open source software practices: The Open Source Initiative Web site [OSI 2007a] lists over 50 different licenses issued by a variety of organizations. The open source movement is still maturing with respect to specific practices.

### Practice Risks

Ineffective use of EAS can lead to schedule breakdowns during initial development as well as throughout the life cycle of a software product line. If a product is unsuitable to begin with or loses vendor support and becomes unsuitable, the result will be a sudden hole in the architecture, with a concomitant hole in the schedule while it is repaired. Major risks can be divided into issues relating to the EAS in question and issues relating to the vendor and vendor policies. Product-related risks include

- **unanticipated interactions:** Unanticipated interactions may occur between the EAS and other components, resulting in a system that does not behave as intended. The mitigation to this risk is to study the EAS and its interactions thoroughly and make specific tests during integration testing. (See the "Testing" practice area.)
- **poor EAS quality:** Similar to the previous risk, EAS without extensive track records pose the risk of failing to meet the reliability standards for the system. Failure to adequately qualify or test EAS can admit an unacceptable product into the system.
- **inappropriate product for the job:** Failure to comprehensively qualify EAS for its intended use may result in the selection of EAS that fails to meet the behavioral requirements or (more likely) the quality requirements such as performance and security.
- **lack of adaptability:** EAS often needs to be adapted to fit within an architecture and environment. Since EAS is usually packaged as a black box, this adaptation may be costly and result in

unpredictable behavior, because the internal workings and assumptions of the EAS are unknown. Changing (or persuading the vendor to change) EAS is a risky move in any case, because the result will be an untested one-of-a-kind piece of software that won't track with future releases or other products in the vendor's inventory.

- **replacements for EAS:** Replacements or substitutions for EAS will not represent one-for-one substitutions. Sometimes new EAS will be available that will perform most of the functions of an existing product. On the other hand, EAS may overlap partially with several components. Decisions about replacements need to be made within the context of the evaluation criteria for the selection of EAS.

Vendor-related risks include

- **inopportune updates to EAS:** EAS updates will almost certainly not be synchronized with updates to the product line, and they may conflict with the technical direction of the product line. Make sure to develop an update strategy for making decisions about when to update EAS, as well as when to forego updates.
- **lack of support for EAS:** The support for EAS will vary by product and vendor. Some products will be withdrawn by a vendor, and some vendors may go out of business. It is important to include specific evaluation criteria based on the vendor's stability, product strategy, support record, and update policy for specific EAS.

## Further Reading

[Clements 2002c, p. 96]
Clements and Northrop provide an example of COTS product evaluation criteria.

[FSF 2007a] Likewise, the Free Software Foundation (www.fsf.org) discusses free software.

[Kenwood 2001a] Kenwood's paper provides a business case study of open source software conducted by the MITRE Corporation for the U.S. Army.

[OSI 2007a] The Open Source Initiative (www.opensource.org) is a good resource for learning more about using open source software.

[SEI 2007c] The SEI's CBS Web site provides a host of papers and monographs on COTS product usage and pointers to further resources, including courseware. The "Gotchas of COTS for Acquisition Managers: What You Don't Know Can Hurt You" presentation (available at http://www.sei.cmu.edu/cbs/cbs_slides/stc98/Gotchas/index.htm) is a good myth-shattering introduction to the issues.

[Wallnau 2002a] The book by Wallnau, Hissam, and Seacord is a comprehensive treatment of the integration of commercial components in a system.

[Wheeler 2005a] Wheeler's paper covers a range of issues associated with using EAS.

## Technical Management Practice Areas

Technical management practices are those management practices that are necessary for the development and evolution of both core assets and products. They are

- configuration management
- make/buy/mine/commission analysis
- measurement and tracking
- process discipline
- scoping
- technical planning
- technical risk management
- tool support

### Configuration Management

> *The purpose of Software Configuration Management is to establish and maintain the integrity of the products of the software project throughout the project's software life cycle. Software Configuration Management involves identifying configuration items for the software project, controlling these configuration items and changes to them, and recording and reporting status and change activity for these configuration items [SEI 2000a].*

Configuration management (CM) refers to a discipline for evaluating, coordinating, approving or disapproving, and implementing changes in artifacts that are used to construct and maintain software systems. An artifact may be a piece of hardware or software or documentation. CM enables the management of artifacts from the initial concept through design, implementation, testing, baselining, building, release, and maintenance.

At its heart, CM is intended to eliminate the confusion and error brought about by the existence of different versions of artifacts. Artifact change is a fact of life: plan for it or plan to be overwhelmed by it. Changes are made to correct errors, provide enhancements, or simply reflect the evolutionary refinement of product definition. CM is about keeping the inevitable change under control. Without a well-enforced CM process, different team members (possibly at different sites) can use different versions of artifacts unintentionally; individuals can create versions without the proper authority; and the wrong version of an artifact can be used inadvertently. Successful CM requires a well-defined and institutionalized set of policies and standards that clearly define

- the set of artifacts (configuration items) under the jurisdiction of CM
- how artifacts are named
- how artifacts enter and leave the controlled set
- how an artifact under CM is allowed to change
- how different versions of an artifact under CM are made available and under what conditions each one can be used

- how CM tools are used to enable and enforce CM

These policies and standards are documented in a CM plan that informs everyone in the organization just how CM is carried out.

### Aspects Peculiar to Product Lines

CM is, of course, an integral part of any software development activity, but it takes on a special significance in the product line context. As illustrated in Figure 8, CM for product lines is generally viewed as a multidimensional version of the CM problem for one-of-a-kind systems.

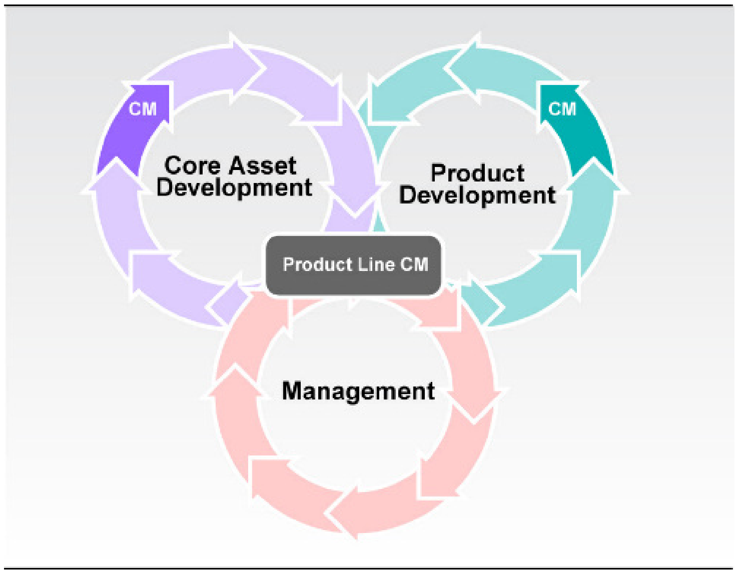


*Figure 8: Configuration Management and Software Product Lines*

The core assets constitute a configuration that needs to be managed, each product in the product line constitutes a configuration that must be managed, and the management of all these configurations must be coordinated under a single process.

CM for product lines is therefore more complex than it is for single systems. CM capabilities such as parallel development, distributed engineering, build and release management, change management, configuration and workspace management, and process management must be supported by the tools, processes, and environments put in place for CM in a product line context. In particular

- In single-system CM, each version of the system has a configuration associated with it that defines the versions of the configuration items that went into that system's production. In product line CM, a configuration must be maintained for each version *of each product*.
- In single-system CM, each product with all of its versions may be managed separately. In product line CM, such management is untenable, because the core assets are used across all products. Hence, the entire product line is usually managed with a single, unified CM process.
- Product line CM must control the configuration of the core asset base and its use by all product developers. It must account for the fact that core assets are usually produced by one team and

used in parallel by several others. Single-system CM has no such burden: the component developers and product developers are the same people.

- Only the most capable CM tools can be used in a product line effort. Many tools that are adequate for single-system CM are simply not sufficiently robust to handle the demands of product line CM. (See the “Tool Support” practice area for a more complete discussion of tools.)

The mission of product line CM is allowing the rapid reconstruction of any product version that may have been built using various versions of the core assets and development/operating environment *plus* various versions of product-specific artifacts. One product line manager summed up the problem this way: “Sometime, in the middle of the night, one of your customers is going to call you and tell you that his version of one of your products doesn’t work. You are going to have to duplicate that product in your test lab before you can begin to troubleshoot.”

Product line CM must also support the process of merging results either because new versions of core assets are included in a product or because product-specific results are introduced into the core asset base. Finally, since introducing changes may affect multiple versions of multiple products, you’ll want your product line CM system to deliver sound data for an impact analysis to help you understand what impact a proposed change will have.

Although few papers or reports on CM mention software product lines explicitly, academic research and industry analysis show that software CM (SCM) is clearly evolving in ways that will support the needs of product lines. Westfechtel and Conradi, observing the overlap between software architecture and SCM, describe five approaches for the integration of the two disciplines [Westfechtel 2003a]. Estublier and colleagues document both successful (e.g., change sets) and failed (e.g., advanced system models) transitions of research into industrial practice and observe that current research is breaking the underlying assumption that SCM is programming-language and application independent. In particular, they note that all high-end SCM systems are slowly but surely covering the spectrum of functionality identified by Dart in 1991 [Estublier 2005a, Dart 1991a].

Similarly, on the industry side, Schwaber at Forrester Research observes that today’s SCM market encompasses a range of four solution segments of cumulatively increasing functionality: (1) version control, (2) software configuration management, (3) process-centric software configuration management, and (4) application life-cycle management [Schwaber 2005a]. According to Schwaber and colleagues, all the major vendors now offer process-centric SCM solutions, and there is growing interest in expanding SCM into application life-cycle management.

## Application to Core Asset Development

The entire core asset base is under CM, with support for all the tasks described in the preceding section. Core assets, after all, may be developed in parallel by distributed teams, may need their builds and releases to be managed, and so forth. Beyond this support, however, core asset development requires other features of the organization’s CM capability. First, it requires a flexible concept of assets that encompasses hardware, software, and documents of all varieties. One of the more useful features of a CM tool is the ability to report the differences between two versions of an artifact. However, doing so often requires fluency in the language in which the artifact is represented. (If you’ve ever tried to execute a DIFF command on two binary files, you get the point.) Thus, a tool's difference-reporting capabilities may weigh heavily in the selection process.

### Application to Product Development

The CM process should make it easy to set up the initial configuration of a new product. Each time a new product is developed (which can occur very often in a healthy and robust product line), the task of determining the appropriate core assets and how to make them available must be supported.

CM also has to keep track of all the versions of configuration items that are used, including the version of the tool environment (compilers, test suites, and so on) used to create the configuration. Incorporating new versions of core assets to build a new version of a product is a task that requires an impact analysis that must be supported by CM. Changes in core assets must be communicated via the CM process to the core asset development organization.

### Example Practices

**IEEE/ANSI standard for CM plans:** There is a finely detailed IEEE/ANSI CM standard that contains a comprehensive outline for a CM plan as well as several fully worked out examples of CM plans for different kinds of systems and organizations [IEEE 1987a]. These plans contain change control policies, describe organizational roles, define artifact life cycles, and, in general, make a fine starting point for an organization wishing to craft its own CM plan. One sample plan, called the "Software Configuration Management Plan for a Product Line System," is of particular interest. This plan is for a hypothetical organization that produces many versions of a product for a multitude of customers, some of whom are internal to the organization itself. This plan hypothesizes an engineering group (that would seem to mirror our notion of a core asset group) as well as several product groups. This and other similarities to our concept of a product line built as a product family make this plan an excellent place to start.

**CMMI steps for CM:** SEI Capability Maturity Model Integration, Version 1.1 for Systems Engineering and Software Engineering (CMMI-SE/SW, V1.1) lists the following practices as instrumental for a CM capability in an organization [SEI 2000a]:

1. Identify the configuration items, components, and related work products that will be placed under configuration management.
2. Establish and maintain a configuration management and change management system for controlling work products.
3. Create or release baselines for internal use and for delivery to the customer.
4. Track change requests for the configuration items.
5. Control changes in the content of configuration items.
6. Establish and maintain records describing configuration items.
7. Perform configuration audits to maintain the integrity of the configuration baselines.

**Best practices from practitioners:** Wingerd and Seiwald have written a set of high-level best practices based on their experience, and the experience of others, with deploying SCM [Wingerd 2005a].

Berczuk builds on the success of the “patterns” movement to cast best practices as a collection of 15 SCM patterns [Berczuk 2003a]. Walrad and Strom propose an alternative to the customary branch-by-release branching model [Walrad 2002a], and the Configuration in Industrial Product Families (ConIPF) program in Europe applies the concept of configuration models from knowledge engineering to the problem of product derivation in product families [Hotz 2006a].

## Practice Risks

CM imposes intellectual control over the otherwise unmanageable activities involved in updating and using a multitude of versions of a multitude of artifacts, both core assets (of all kinds) and product-specific resources. Without an adequate CM process in place, and without adequate, adherence to that process, developers will not be able to build and deploy products correctly, let alone recreate earlier versions of products. Inadequate CM control can result from the following:

- **an insufficiently robust process:** CM for product lines is more complex than CM for single systems. If an organization does not define a robust enough CM process, CM will fail, and the product line approach to product building will become less efficient.
- **CM occurring too late:** If the organization developing the product line does not have CM practices in place well before the first product is shipped, building new product versions or rebuilding shipped versions will be very time-consuming and expensive, negating one of the chief benefits of product lines.
- **multiple core asset evolution paths:** There is a risk that a core asset may evolve in different directions—something that can happen either (1) by design in order to enable the usage of a core asset in different environments such as multiple operating systems or (2) by accident when a core asset evolves within a specific product. When done by design, the evolution might be unavoidable and increase the complexity of the CM. You must watch for evolution by accident, because it can degrade the usefulness of the core asset base.
- **unenforced CM practices:** Owing to the complexity of the total product line configuration, not enforcing a CM process can result in total chaos (a result that’s much worse than that for a single-system).
- **insufficiently robust tool support:** CM that is sophisticated enough to support a nontrivial product line requires tool support, and there is no shortage of available commercial CM systems. However, most of them do not directly support the functionality required for the CM to be useful in a product line context. Many of them can be “convinced” to provide the necessary functionality, but this convincing is a time-consuming task requiring specialized knowledge. If the organization fails to assign someone to customize the CM system for the product line, the CM tool support is likely to be ineffectual. Such a person needs to have both a good understanding of the product line processes and a solid grounding in CM.

## Further Reading

[Burrows 2005a]
Burrows and Wesley provide a thorough comparison of commercially available CM tools. Their work is required reading for anybody who plans to buy such a tool. Relevant CM industrial standards from

IEEE and ISO should also be on every project manager's bookshelf [IEEE 1987a, IEEE 2005a, ISO 1995b].

[Crossroads 2006a]
The CM Crossroads Web site (www.cmcrossroads.com) is an online resource that offers articles, white papers, book reviews, tool reports, discussion forums, and the *Configuration Management Journal*. It also includes links to other sites with CM information; *Software Development* magazine's July 2005 article on CM tools and trends is one such example. (To locate that article or others, go to http://www.sdmagazine.com.[6])

[Krueger 2002a]
Krueger provides an in-depth discussion of variation management for software product lines.

[Leon 2005a]
Leon's book, now in its second edition, is a comprehensive general CM reference.

## Make/Buy/Mine/Commission Analysis

In "A Note on Terminology," we pointed out that software enters an organization in one of three ways: it can be

- *built* in-house
- *purchased* from a commercial vendor—either whole (as in a commercial off-the-shelf [COTS] component) or as licensed rights to use the software (as in open source software or a Web-based service)
- *commissioned* through a third party to be built especially for the organization

Software that is built in-house can actually be constructed anew or *mined* from software already in the organization for use in a new effort. Every piece of software that is part of a development effort arrived as the result of an all-encompassing four-way choice that we call "make/buy/mine/commission." Organizations that build software systems must all make this choice, but they usually lack a conscious rationale for their selection. The purpose of this practice area is to both underscore the necessity of making a conscious and reasoned choice and describe some of the analyses that can help an organization make the right choice.

Techniques from the discipline of decision analysis apply well here. Decision analysis is the process of applying analytical methods to decision making in situations where there is uncertainty, multiple conflicting objectives, or dynamic change [Clemen 1991a, Hammond 1999a]. Such a situation exists when an organization is trying to choose the best way to obtain software and must, as a result, weigh several business, technical, and political factors. In such a case, making the decision requires both qualitative and quantitative analysis.

[6] In May 2006, *Software Development* magazine merged with *Dr. Dobb's Journal*. However, past issues of the magazine are still available at this URL.

Sometimes the decision is straightforward. For some organizations, all the software assets are developed in-house for proprietary (or political) reasons. For other organizations, all the assets are commissioned because of organizational policy or because of unique requirements and a lack of in-house development resources. (In the U.S., most government organizations, such as the DoD, fit into this category.) If no other organization but yours has the necessary domain expertise in a component's realm, "buying" and "commissioning" are not viable alternatives. Conversely, if you have neither the skill nor the history to build a component, "making" and "mining" are not going to work. More commonly, however, some of the software will be built from scratch, some will be mined, some will be purchased on the open market, and some will be commissioned.

The make/buy/mine/commission decision for software is based on strategic factors such as the cost, schedule, staff availability, and expected quality and fitness of purpose that each alternative offers. For example, mining is valuable only when a project that uses the mined assets can be completed on time at a lower cost and produce a capability equal to or greater than that of comparable assets obtained through other means. Any calculation of reuse cost should include the total cost of use over the lifetime of the new product or products, not just the cost of mining/restoring a particular set of assets. In practice, improvements on one of the scales (at a cost to the others) may produce a significant tactical advantage. For example, if mining and restoration gain time but lose cost and functionality (relative to building from scratch), they could still provide a significant advantage if time to market is a primary driver. The direct cost of each alternative is a factor, but so is the opportunity cost: what could your staff be doing instead of building a component if you buy or commission it? If you developed it in-house, what could you do with the funds you would save by not employing an expensive contractor? If the staff is underused, the "make" and "mine" options get more weight. If the staff is overused and the schedule is tight, the "buy" and "commission" options get more weight. Other factors influence the decision when a service-oriented approach is used. For example, are suitable service providers available to offer the necessary types of services in a form that is compatible with the product line architecture? Will it be possible to sign a service level agreement (SLA) with particular vendors to guarantee the quality of service (QoS) needed for the product line?

When analyzing the

- "buy" option, refer to the "Using Externally Available Software" practice area
- "make" and "mine" options, practices in the "Architecture Evaluation, "Mining Existing Assets," and "Component Development" practice areas will help
- "commission" option, you must rely on the contractor's past performance for insight into whether it will be able to deliver as promised. Here, vendor reliability and stability are key, as for a COTS component; see the "Developing an Acquisition Strategy" practice area.

The make/buy/mine/commission decision for software should also be based on a frank assessment of an organization's own capabilities. Establishing working relationships with contractors can be tricky, and the legalities of drawing up an ironclad contract can be formidable for those without experience. If your organization does not have a standing legal department for which such relationships are pro forma, you might be better off on the "make/mine" side of the equation—that is, unless building such a capability for the future makes strategic sense for your organization. Conversely, if your development capabilities are weak, letting others tackle the technical complexities may be the best approach—

again, unless you are trying to improve your organization's development skills or domain expertise. In either case, start your venture into the unknown with small steps, using the newest approaches for only the least critical components. In addition, have contingency plans to handle major missteps along the way.

Also keep in mind that the four options are not always mutually exclusive-for a given asset, you might do a little of each; for example

- A component can be built largely from scratch but with some percentage derived from a legacy system.
- Commissioned software is sometimes based at least partially on legacy assets.
- How much customization or alteration of a COTS system is allowed before that system falls into one of the other three categories?
- A software component could be purchased as a service on an ongoing basis from a service provider.

Since the make/buy/mine/commission decision analysis can be complex, making the decision consciously and documenting the rationale is a good idea.

### Aspects Peculiar to Product Lines

The analysis approach for a product line is similar to that for single systems but with two main differences. When a product line is involved, you must perform the analysis (1) while constantly considering the satisfaction of the product line's production constraints and strategy and (2) using different weighting factors because of the reasons described below:

- Costlier options that would be ruled out for single systems may be acceptable for product lines because the cost can be amortized over a number of products. For example, in a single system, you might be willing to use a COTS component because it would be cheaper than building your own. However, in a product line, you may be willing to pay the higher cost of in-house development, so your entire group of products is not held hostage by a vendor's version release and upgrade schedule. Similarly, certain licensing arrangements might make the use of externally available software reasonable for a single system but prohibitively expensive for a product line.
- The "make," "mine," and "commission" options are usually more expensive because if the asset is destined to join the core asset base, it will have to be robust enough to be reusable across the entire family of products.
- The "buy" option, on the other hand, may not be any more expensive than for a single-system case (licensing agreements notwithstanding) because COTS components tend to be built for generic usage. Of course, you still have to find an off-the-shelf component in the commercial marketplace that has the required variability and quality. Searching for and qualifying a component that's suitable for your product line may be more expensive than if it were going to be used in only one system.
- Considering external services (provided by outside organizations) as product line assets impacts the "buy" option because what you're buying is the right for your system to use those services.

You will have to determine how many external services are to be used, the ongoing costs involved, and whether some of those costs will be passed on to the end users.

- Product lines tend to be built on a legacy foundation; the realization that a company is building many similar products is often the impetus for the product line approach. Hence, “mining” is a more viable alternative in a product line because of the likely existence of a rich legacy base.

The decision analysis for a product line must also look further into the future for multiple products that will be spawned, each having its own lifetime. In a product line, much more is riding on the decisions about how and where to obtain software. Analysis is essential, and rigorous analysis is warranted.

### Application to Core Asset Development

Because all core assets have to come from somewhere, make/buy/mine/commission analysis is at the root of core asset development, and the analysis applies to non-software assets as well as software assets. The possibility of commissioning entire swaths of product line development (such as domain analysis, scoping analysis, market analysis, requirements engineering, and testing) should be considered. Requirements, architectures, and designs—as we describe in the “Mining Existing Assets” practice area—are superb candidates for recovery and rehabilitation from previous systems. The criteria to consider during the analysis include

- quality
- the cost (including the opportunity cost, that is, the loss in terms of the productive work the staff could have been doing instead)
- alignment with the product line requirements
- alignment with the product line architecture
- sufficient flexibility for supporting requisite variation among the products in the product line
- maintainability
- the schedule
- the ability of your organization to prosecute each of the four options successfully
- the availability of core assets as services

The source of the product line architecture must be chosen prior to, or at least concurrent with, deciding how to obtain individual component or service assets. The architecture, by definition, specifies constraints for component and service assets-constraints that must be factored into any analysis about how those assets will be obtained.

### Application to Product Development

Individual products in the product line may need additional software not contained in the core asset base. The same decision analysis process should be used to determine how to obtain this software. The criteria may be less demanding if the component or service in question will not be a candidate for inclusion in the core asset base. Although the component or service would still need to align with the product line architecture, it wouldn’t have to offer the variability needed to support other products. On the other hand, the cost of obtaining the component or service may be more of a consideration because it has to be absorbed by a single product rather than being amortized over a family of products.

For example, if there isn't a human-computer interface (HCI) component or subsystem in the core asset base and a specific product needs one, the product builder can select the HCI component that is optimal for the needs of that particular product and the allowable budget. On the other hand, if an HCI component would be a viable candidate for addition to the core asset base, the decision analysis would rely on the same criteria used for core asset development, as described above. Furthermore, if external services are to be used within an individual product, the cost of using these services must be considered.

## Example Practices

**Breadth-first analysis of options:** One approach is to adopt a breadth-first strategy in which each of the four options for obtaining software is considered and explored before any is eliminated. For some options, the consideration may be trivial. For example, if organizational policy dictates the commissioning of all software systems, the "make" choice requires no consideration: it's simply not an option.

Example questions for analyzing the make/buy/mine/commission decision are given below. First, there is an umbrella set of questions for the breadth-first strategy, followed by four sets of specific questions—one for each of the four options. Since each organization is unique, these sample questions should be viewed as a starter set that you can customize and augment as necessary.

**Umbrella questions that follow the breadth-first strategy:**

- What are the time constraints for obtaining the asset? What external factors drive them? How reliable are the external predictions?
- How well-defined is the asset that must be obtained? Have the product line requirements been defined? Have the required software commonalities and variabilities of all the products in the product line been defined? Has the system architecture into which the software will fit been defined completely? (If any of these items are not well-defined, it will be hard to hand off the responsibility for developing the asset, either software or non-software, to an outside party.)
- Is there a market for the asset that is separate from the market for the product line? (If not, you probably won't find the asset on the open market as a COTS product.)
- How flexible are the product line requirements and to what extent might the availability of a specific asset impact the requirements?

After this initial analysis, the choices may be narrowed to two, three, or even one of the four major options. A more detailed analysis for each option still in the running could include questions such as the following:

**Sample questions for the "make" option:**

- Are developers with the necessary expertise available?
- How would those developers be used if they were not on this project?

- If additional personnel need to be hired, will they be available within the needed time frame?
- How successful has the organization been in developing similar assets?
- What specific flexibilities are gained by developing products in-house as opposed to purchasing them?
- What development tools and environments are available? Are they suitable? How skilled is the targeted workforce in their use?
- What are the costs of development tools and training, if needed?
- What are the other specific costs of developing the asset in-house?
- What are the other specific benefits of developing the asset in-house?

**Sample questions for the "buy" option:**

- What assets are externally available either as COTS software, open source software, freeware, or services?
- How well does the externally available software conform to the product line architecture?
- How closely does the externally available software satisfy the product line requirements?
- Is it possible to establish an SLA with the vendors of the software services that will meet the needs of the product line requirements?
- Are small changes in the externally available software a viable option?
- Is source code available with the software? What documentation comes with the software?
- What are the integration challenges?
- What, if any, redistribution rights are purchased with the externally available software?
- What other specific costs are associated with purchasing the software? Is there an ongoing cost for the use of the software services?
- What other specific benefits are associated with purchasing the software or the service?
- How stable are the vendors?
- How often are upgrades produced? How relevant are the upgrades to the product line?
- How responsive is the vendor to user requests for improvements?
- How strong is the vendor's technical support for the software or service?

**Sample questions for the "mine" option:**

- What legacy systems are available to mine assets from?
- How close is the functionality of the legacy software to the required functionality?
- What is the defect track record for the software and non-software assets?
- How well are the legacy systems documented?
- What mining strategies are appropriate? How expensive are they?
- What experience does the organization have in mining assets?
- What mining tools are available? Are they appropriate? How skilled is the workforce in their use?

- What are the costs of mining tools and training, if needed?
- What other specific costs are associated with mining the asset?
- What other specific benefits are associated with mining the asset?
- What changes must be made to the legacy asset during the mining? What are their costs and risks?
- What types of non-code assets are available to be mined? What, if any, modifications or additions will they need?

**Sample questions for the "commission" option:**

- What contractors are available to develop the asset?
- What is the track record of the contractor in terms of schedule and budget?
- Is the acquiring organization skilled in supervising contracted work?
- Are the requirements defined well enough to subcontract the asset's development?
- Are interface specifications well-defined and stable?
- What experience does the contractor have with the principles of product line development?
- Who needs to own the asset? Who maintains it?
- What other specific costs are associated with commissioning the asset?
- What other specific benefits are associated with commissioning the asset?
- What are the costs of maintaining the commissioned asset?
- Does commissioning the asset involve divulging to the contractor any technology or information that, in the acquiring organization's best interest, should not be revealed?

These question sets are meant to be starter sets. Weigh these factors according to your organization's needs and experience, and augment them with factors of your own.

**OAR/SMART:** The SEI Options Analysis for Reengineering (OAR) approach is used to make decisions on mining assets for product lines and to address the specific mining questions outlined above [Smith 2002a]. OAR is explained in more detail in the "Mining Existing Assets" practice area. Similarly, the SEI Service-Oriented Migration and Reuse Technique (SMART) helps organizations analyze a legacy system to determine whether its functionality, or subsets of it, can be reasonably exposed as services in a service-oriented architecture (SOA) [Lewis 2005a]. This technique may be useful if a service-oriented approach is used within a product line. Economic models such as SEI SIMPLE (discussed in the example practices for the "Building a Business Case" practice area) can help answer questions regarding the relative costs or cost savings of each option.

In some cases, the "commission" option may involve commissioning a supplier to mine assets from an existing system. Bergey and colleagues suggest that under these circumstances a specific set of practices should be included to

- determine that potential suppliers have the needed skills, mining experience, and tool support
- ensure that mining recommendations are based on solid technical analysis

- conduct a mining exercise to validate some aspect of the supplier's mining process and ability to produce product line assets
- secure visibility into the planning, mining analysis, and decision-making process of a potential supplier [Bergey 2004a]

**Tool support:** Although spreadsheets are the most common tool used in decision analysis, tools that are more sophisticated are also available on personal computers. For example, TreeAge Software offers tools called DATA and DATA Interactive [TreeAge 1999a]. With DATA, an analyst can build sophisticated decision models as decision trees, influence diagrams, or Markov processes and then analyze them using analysis tools such as sensitivity analysis, cohort simulation, and Monte Carlo simulation. A companion product, DATA Interactive, provides a model customization and delivery system for accessing and analyzing the decision models created in DATA. These tools require knowledge about decision analysis principles, the problem domain, and the tools' functionality for building and analyzing models.

### Practice Risks

Do not confuse the risks of this practice area with those of the practice areas associated with each of the four options (i.e., "Component Development," "Using Externally Available Software," "Mining Existing Assets," and "Developing an Acquisition Strategy"). The risks of this practice area are related to choosing among those options.

The major risk specific to this practice area is, of course, choosing the *wrong* option, which can result in inappropriate assets and undue direct (or opportunity) costs. This risk could occur for the following reasons:

- There are not enough data on which to base decisions.
- The data are inaccurate.
- Political pressures force the decision.
- There is no process for the decision analysis.
- Schedule pressure precludes conducting a thorough analysis.
- Not all the options are considered. Normally, all four options should be kept on the table for the initial analysis steps.
- There is no established or documented product line architecture to factor into the decision.
- There are poor estimates of the probabilities of projected outcomes.
- The product line requirements have not been defined adequately.
- Hidden interactions between alternative choices exist and may produce unpredictable consequences.
- There is a lack of documentation to support decision analysis.
- The differences in business interests between the acquiring organization and the supplier are not adequately factored into the decision-making process when choosing the commission option.

### Further Reading

[Clemen 1991a] & [Hammond 1999a]
Clemen, Hammond, Keeney, and Raiffa provide excellent background reading in decision analysis, which is necessary for applying the "Make/Buy/Mine/Commission" practice area successfully.

## Measurement and Tracking

Software development efforts must be tracked to ensure that they meet their organizational goals. While informal and qualitative tracking techniques are valuable, they should be supplemented with objective and quantitative measurement techniques.

The purpose of measurement is to support project tracking and guide management decision making [Grady 1992a, Park 1996a]. The manager sets goals, defines objectives that satisfy those goals, and then creates a plan and applies resources to achieve those objectives. In order to determine whether the goals are being achieved as time passes (that is, whether the plan is working), the manager must have data that indicate the state of the effort. By tracking and analyzing relevant measurable attributes of the effort's process and product, the manager has a quantitative window on the progress toward the effort's goals. The manager can also detect issues that indicate when the effort has diverged from expectations. The manager can then revise the effort's goals, plan, or resources to address these issues—or recalibrate everyone's expectations.

Measurement-based project tracking is based on

- defining and refining the project goals that measurement will help track
- identifying the success criteria and indicators for each of those goals
- defining the appropriate measures
- developing a plan to operationalize and verify those measures

Once the plan is established, adherence to it should be monitored continually to ensure that the measures are being used effectively. The plan should also be revised as the organization's and effort's goals evolve.

### Aspects Peculiar to Product Lines

The techniques for collecting and tracking data are the same for a product line as for a single system; both situations require an initiation phase to plan for the measurements and a performance phase to collect data and analyze results. However, in a product line, data collection must provide information from three perspectives, not just the single perspective of product development. Recall the three essential activities of product line development:

1. core asset development, comprising the efforts needed to produce reusable assets and the supporting infrastructure for their use
2. product development, comprising the efforts needed to produce individual products for customers

3. management of the overall product line, including the strategic planning and direction of a total product line enterprise

Appropriate measures must be collected and tracked to support each of these activities. A product line manager and/or a product line management oversight group is concerned with tracking whether the overall multiproduct effort is efficient, effective, and progressing properly toward achieving its strategic goals and satisfying the product line's production constraints (see "Core Asset Development"). Managers of core asset development are concerned with the quality and usefulness of the core assets they produce and the productivity of the people who produce them. Individual product managers are concerned with the efficiency of their staff and the quality of their products.

These differing concerns are complementary in that the measures required to track the progress of the overall product line effort are mostly aggregated from the measures required to track the progress of its constituent core asset and product efforts. For example, the product line goal Better Quality (which might be indicated by fewer errors after delivery) across all products is tracked in terms of

- the level of quality (number of errors after delivery) determined in each individual product
- how those levels compare and change over time

The quality level of the individual products is influenced heavily by the quality of the core assets from which they are built. So having fewer errors in core assets after their delivery to the product developers becomes a local goal of core asset development and is also tracked.

As another example, consider the product line goal Increased Profitability of Product Development. The profitability is tracked by the measures reported while component core assets are assembled according to the production plan. The cost measures associated with the development and evolution of the core assets also need to be factored into the profitability measurement.

Over and above the specifics of which measures to track and which data to collect for software product lines, this practice area has another dimension that makes it highly relevant. The ability of an organization to collect and analyze data and to track measures about itself *at all* says a lot about the organization: it is comfortable with disciplined processes, accustomed to taking a long-term view, and interested in self-improvement—all of which are the hallmarks of successful product line organizations.

## Application to Core Asset Development

Management overseeing core asset development has two concerns: (1) the efficiency of the core asset effort and (2) its effectiveness in benefiting the associated product efforts that are its clients.

To meet efficiency goals, management should focus on tracking the cost and time required to develop the core assets. Satisfying efficiency goals means that core assets will have a minimum investment cost (therefore, requiring a lower payback from product efforts) and be available sooner for product efforts to use. These benefits mean that the overall product line effort will spend less on both core assets and products, permitting it to either lower prices or increase profits as the company's business strategy and market conditions dictate.

To meet effectiveness goals, the manager should focus on providing the core assets that offer the best chance of reducing the amount work needed in product efforts. Table 4 tells what such effectiveness measures can indicate and help determine.

*Table 4: Effectiveness Measures*

| **Measures for effectiveness indicate** | **Which helps to determine** |
|---|---|
| which core assets are used by product efforts and how often | whether the available core assets are useful |
| how many bugs are found in core assets by the product developers | the quality of the core assets |
| how much product efforts expend in finding, tailoring, and integrating assets | needed improvements in supporting infrastructure |
| where product efforts spend time otherwise | opportunities for future core asset or infrastructure work |

## Application to Product Development

The measurement activity for product development efforts has two facets: one that's conventional and one that's product line specific. The conventional facet involves the collection and analysis of measures needed for the management of any product development effort. For example, managers of product development efforts in general need to track the time and cost of their work activities, the quality of the products developed, and customer satisfaction. These particular measures are applicable to both products developed from core assets and those that are not.

The facet specific to product lines involves the collection of measures that product line and core asset development managers need in order to be effective managers, but those measures can only come from product development. The ones needed for the management of core asset development relate to the quality and usability of core assets from the perspective of the product efforts. Included are measures such as which core assets were used in which products, the amount of effort required to incorporate a core asset into a product, and the number of problems encountered. The measures needed for product line management are generally a subset of those needed to manage core asset and product efforts, but they are aggregated across all the product line's efforts. Product line management will use these measures to determine whether resources are allocated properly between core asset and product efforts and whether these efforts are achieving the intended market results for the business.

## Example Practices

**Goal-driven software measurement (GDSM):** GDSM [Park 1996a] is a ten-step technique that supports project tracking by defining the appropriate project measures based on the project goals and developing a plan to operationalize and verify those measures. GDSM is based on the earlier work of Basili and Weiss [Basili 1984a]. The ten steps of GSDM are

1. Define the goals.
2. Refine the goals using clarifying questions.
3. Define subgoals for the relevant stakeholders.

4. Operationalize the goal statements.[7]
5. Determine the success criteria for the stakeholder.
6. Determine the success, progress, and analysis indicators associated with those criteria.
7. Define the organizational strategies and activities for achieving their goals.
8. Determine the measures and data elements needed by all identified indicators.
9. Assess the current infrastructure and identify actions needed to implement the measures.
10. Develop a measurement plan that addresses the actions to be taken and how the measurements will be verified.

**Choosing measures:** A good starting point for choosing product line measures comes from the work of Zubrow and Chastek [Zubrow 2003a]. They describe indicators that are of interest to a product line manager, a core asset development manager, and a product development manager (see Table 5). Under each category, a broad set of measures (listed in the "Goal" column of the table) is defined that returns information about performance (measuring cost, schedule, and quality of product efforts), compliance (measuring the adherence of the product line effort to established procedures and processes), and effectiveness (characterizing how the overall product line effort is meeting its goals).

*Table 5: Product Line Indicators and Measures [Zubrow 2003a]*

| Goal | Product Line Manager | Core Asset Development Manager | Product Development Manager |
|---|---|---|---|
| Improved Performance | • total product development cost<br>• productivity<br>• schedule deviation<br>• time to market<br>• trends in defect density<br>• number of products (past, current, future)<br>• time spent on life-cycle activities | • cost to produce core assets<br>• cost to produce infrastructure<br>• schedule deviation<br>• defect density in core assets<br>• number and type of artifacts in asset library | • direct product cost<br>• defect density in application artifacts<br>• percent reuse |
| Compliance | • mission focus<br>• process compliance | • mission focus<br>• process compliance | • process compliance |
| Increased Effectiveness | • return on investment<br>• market satisfaction | • core assets utility<br>• core assets cost of use<br>• percent reuse | • customer satisfaction |

"Building a Business Case: Example Practices" discusses specific measures for supporting a business case.

**Collecting data:** Different types of measures require different data collection techniques. Common techniques include

[7] This step involves determining what properties and attributes will be analyzed and why, the environment and context of the collected data, who will use that data, and who will use the results.

- the direct measurement of the observable attributes of a process or product, such as the date that a baseline was created or the size of a core asset
- the indirect measurement of objective attributes, such as the time it took a developer to create a core asset or a work product
- surveys for the measurement of subjective attributes, such as how easy or pleasant an automated product is to use (This type of information helps measure customer satisfaction with a product line.)
- the derivation of implicit attribute measures as computations from other measures, such as (1) the cost of a work product as a function of its size and the cost of the developer's time or (2) the usability of an automated product as a function of the number of user mistakes

Each of these techniques can be applied manually or with the assistance of automated tools, depending on the size of the effort being undertaken and whether time or money is the more limited resource.

**Reuse measures:** A higher level of software reuse is not, in itself, an end goal of a product line effort but merely a strategy for achieving goals such as shorter time to market. However, because software reuse is such a cornerstone of the product line strategy, it is a useful quantity to measure. Beyond that, it is useful to have a measure-based model of what the reuse is buying you, in terms of cost avoidance, return on investment, or some other goal-related quantity. Poulin provides several such models [Poulin 1997a]. Some of the proposed models also have associated tools that support the appropriate analyses and presentation of results.

## Practice Risks

A poor data collection and measurement program will be a waste of time and resources, will have an opportunity cost (i.e., a loss in terms of the productive work the staff could have been doing instead), and will cause resentment and mistrust of future measurement efforts. It will also fail to inform management of where the organization stands with respect to meeting its product line goals. This failure can, in turn, lead to uninformed management decisions that can undermine the product line efforts and ultimately to an inability to validate the business case that justified that product line effort. Potential causes of inadequate data collection and measures are as follows:

- **measure mismatch:** Measures that are not based on product line, subordinate core asset, or product development goals will result in wasted effort spent collecting data that do not contribute to management decision making.
- **goals without measures:** Goals that have no associated measures will result in managers being unable to detect any issues that hinder the achievement of those goals until an unacceptable expenditure of work or time has been incurred.
- **measurement not aligned:** Any measurement activity that is not integrated into the product line process will result in data collection that
  - does not mesh properly into other product line activities
  - leads to inaccurate results that either hide legitimate issues or raise false issues

- **costly measures:** Measures that are too costly or difficult to obtain will result in failure to track progress, which, in turn, will result in failure or delays in detecting problem issues or interference with the effort's primary work.
- **measures without planned actions:** Measures should track the status of an effort with respect to a specific goal and have associated action plans that can be carried out if the goal is not being met.
- **management by numbers:** Measurement is an important tool for managers, but it is not the end game. Measures can give insight into problem areas and performance, but not everything in an organization is quantifiable. Aircraft pilots are taught that in some circumstances the most important thing they can do is to disregard their instincts and fly solely by what their instruments are telling them. While that is good advice for pilots, it isn't appropriate for managers who must complement measures with experience and insight in order to understand what is really going on within the organization.

### Further Reading

[Geppert 2003a]
Geppert and Weiss describe Avaya Labs' method for the quantitative assessment of potential product line domains. Their paper defines measures for the potential corporate impact and the likelihood of success when a candidate domain is implemented as a product line.

[Park 1996a]
Park, Goethert, and Florac provide an extensive guide to establishing a measurement activity based on business goals.

[Poulin 1997a]
Poulin treats measurement from the point of view of a reuse organization. His work describes a good suite of reuse-based measures and models.

[Zubrow 2003a]
Zubrow and Chastek suggest a small set of measures to track a software product line. Those measures range from the relatively mature to the not yet validated.

## Process Discipline

The "Process Discipline" practice area is about an organization's capability to define, follow, and improve processes. An essential aspect of software engineering is the discipline required for a group of people to work together cooperatively to solve a common problem. Humphrey defines the term *software process* as "a sequence of steps required to develop or maintain software" [Humphrey 1995a]. Defined processes set the bounds for each person's roles and responsibilities so that the collaboration is successful and efficient. This practice area is about the skills required to plan, define, and execute those various processes successfully.

A first step on the road to process discipline is to determine which processes need to be defined and plan how to do it. Advice on which software processes to enact is provided in various process models. Sheard shows the lineage of several such models [Sheard 1997a]. Widely used examples include

- SEI Capability Maturity Model Integration models (CMMI), which contain essential elements of processes for software development in the categories of project management, engineering, support, and process management [SEI 2007h]
- the International Standards Organization (ISO) 9000 series of quality standards. This set of general standards is applicable to the production of nearly any commodity, with the ISO 9001 standard being particularly pertinent to software development and maintenance [ISO 2007a].
- software development methods that identify which processes to enact, such as agile methods or the Rational Unified Process (RUP). Some agile adherents may prefer to refer to these methods as "shared understandings" or "tacit knowledge" [Boehm 2004b].

Once a particular process model is selected, there are methods for defining it. If an informal (albeit perhaps inconsistent) process exists already, it is useful to document it as a starting point. The SEI Process Change Methodology [Fowler 1999a] and the SEI Personal Software Process [Humphrey 1995a] take this tack. Interviews and stakeholder working sessions are useful in resolving differences about how the process is (or should be) defined. Once the current process is defined, you have a basis for following and improving it. (See the Example Practices in this section.)

An effective process definition is complete enough to get the job done but not so overspecified as to bog things down. At one end of the scale, some agile methods' tacit understandings of process may be adequate for small teams working on relatively simple tasks. For a somewhat more complex task involving more people, an effective process definition might be a simple checklist. For a much more complex task involving many people and specialized knowledge, a much more elaborate process definition may be necessary. While details vary, most process representation schemes require that you identify at least the *activities* that are performed, the *agents* that perform them, and the *work products* that result. The Object Management Group's Software Process Engineering Metamodel (SPEM) provides a more elaborate object-oriented scheme for representing software processes [OMG 2005a].

Ultimately, the process definition should be clear and complete enough to satisfy the following goals:

- Facilitate human understanding and communication.
  - Enable communication about and agreement on the software process.
  - Provide sufficient information to allow an individual or team to perform the intended process.
  - Form a basis for training individuals to follow the intended process.
- Support management.
  - Adapt a general software process to accommodate the attributes of a particular project, such as its product or organizational environment.
  - Support the development of project plans.
  - Monitor, manage, and coordinate the process.

    - Provide a basis for process measurement, such as the definition of measurement points within the context of a specific process.
- Support process improvement.
    - Reuse well-defined and effective software processes on future projects.
    - Estimate the impacts of potential changes in a software process before putting them into actual practice.
    - Assist in the selection of models and tools and their incorporation into a process.

It is management's responsibility to provide support for defining and institutionalizing processes and ensuring that they are followed, coordinated, and improved. Often, management establishes a software engineering process group to carry out the details, while a management steering group provides overall planning and direction. A wealth of information is available on software process management. Zahran provides an introduction to the subject [Zahran 1998a]. Kasse describes process management and improvement that is centered on both the assessment of the current state and on follow-up actions to achieve the desired state [Kasse 2002a]. There are also international conferences and local Software Process Improvement Network (SPIN) chapters throughout the world [SEI 2006c].

## Aspects Peculiar to Product Lines

If software engineering is about a group of people working together to solve a problem cooperatively, product line software engineering requires cooperation in spades. Because of the plurality of products and groups cooperating to develop those products, the entire apparatus will work to its potential only if everyone does his or her job within agreed-upon parameters. For example, no one is allowed to change core assets unilaterally. Core asset developers are not allowed to change them before understanding how the changes will affect the products in (or already out of) the production pipeline. Product developers are certainly not allowed to change them because "clone and own" is a poor form of reuse that is the antithesis of the product line approach. Core assets do indeed change but only as the result of all parties adhering to an agreed-upon process for making any changes.

Besides change control and configuration management other prime examples of product line processes are

- the operational concept for the product line, such as the one embodied in a product line concept of operations (CONOPS) (see the "Operations" practice area). The CONOPS is essentially the expression of an organization's product line enterprise process. Process definition skills must be brought to bear to build an operational definition that can be followed and improved and that will serve the goals of the product line. Process discipline is required to ensure that the operational concept is followed.
- the attached processes that tell product builders how to instantiate core assets for specific products. The attached processes together form the production process, which in turn provides the content for the production plan. The production process must account for a wide variety of core assets and variations and accommodate the different role players who are going to carry out the process.

- the production method that sets the implementation approach for the production plan by specifying the overall set of technologies, tools, and models to be used to produce products. The method provides the guidance core asset builders need to build core assets with appropriate variation mechanisms and their attached processes. The set of core assets and their attached processes must align with the product line's goals and be compatible with each other to form a cohesive, efficiently repeatable production process. The method can be an informally stated set of choices or more formally documented.

Other product line activities also rely on process discipline to achieve their required fidelity. Launching a product line, carrying out technical and organizational planning and risk management activities, collecting and analyzing measures, performing make/buy/mine/commission analysis, and maintaining the correct customer interface all rely on adherence to processes that must be defined and followed. However, not all practice areas require processes of the same rigor. Whereas configuration management (CM) needs an industrial-strength, no-nonsense process that everyone must follow, an exploratory activity such as market analysis might not.

The expertise required to define the above processes varies depending on the type of process being created. The production method requires individuals with considerable technical currency and software design experience. For example, the core asset developers should be involved, since they understand the tools and models currently being used: both are important input to the production method. Those defining a measurement process would instead need to be skilled in defining measures and appropriate data collection and analysis activities.

Because product lines call for the repeated, ongoing, disciplined interaction of separate organizational entities, they rely heavily on the adherence to a process. Process discipline represents an area of expertise that enables many other practice areas to be executed successfully. For this reason, organizations that lack a strong process culture will find deploying a successful product line a perilous proposition [Clements 2002c, Jones 2002a].

## Application to Core Asset Development

Each core asset has an attached process associated with it that explains how the asset is used to build products in the product line. Together, these attached processes form the product line's production process, which in turn is embodied in the production plan that shows how core assets are turned into products. The attached processes differ across core assets. For example, the way in which the product line requirements are used to express those for a specific product differs from the way in which the product line architecture is instantiated to architect the specific product. Since the people who create a core asset are usually not the people who use it, the attached process communicates the asset's intended use. The attached processes all follow the guidance of the production method. Process discipline is the vehicle by which the attached processes and the production method are specified and enforced.

In addition, there are processes that explain how core assets are created, evaluated, maintained, and evolved over the lifetime of the product line. Once again, process discipline is the mechanism by which these processes are defined, followed, and improved.

## Application to Product Development

As part of the production plan, a core asset's attached process is the bridge between the core asset developers and the product developers; it is a way for the core asset builders to speak across any gulf to the product builders and tell them how the core assets should be used to create systems. Therefore, the attached processes have implications for product development as well as for core asset development. Good process definition skills provide easy-to-use, effective, and efficient attached processes; processes that are cumbersome or unwieldy will probably be discarded by product builders in favor of ad hoc, non-repeatable approaches.

Products, like core assets, also have rules for evolution and maintenance that must be captured in a process and then followed.

## Example Practices

Example practices for process discipline include the following.

**Creation and use of electronic process documents:** These documents—which include Web-based process handbooks and electronic process guides—allow process definitions to be focused or presented in different ways to shield the process user from scores of unnecessary details. Instead, the electronic documentation displays a narrowed view that describes at any specific time the process steps that should be performed. Additionally, this approach makes it possible for automatic enforcement and tracking of process execution. In general, role-specific views of a process decrease the complexity of process definition and allow you to focus on those aspects of the process that are relevant to a specific user. Finkelstein, Kramer, and Nuseibeh give a fairly comprehensive overview of various modeling and documentation techniques [Finkelstein 1994a].

**Integrated process support environments:** These environments use computer systems to automate some of the required process steps, such as informing a quality assurance organization that a document is ready for review or sending a change request to core asset developers [Finkelstein 1994a].

**A risk-based approach to choose between adaptive (i.e., agile) and predictive (i.e., plan-driven) process methods:** Boehm and Turner recommend risk analysis that is based on how well a project aligns with a method's "home ground" [Boehm 2004b]. For example, the agile methods' home ground includes the characteristics of a small group of senior developers in an environment of frequent requirements change working on a system of low criticality, whereas the plan-driven methods' home ground includes the characteristics of a large number of junior developers in an environment of infrequent requirements change working on a system of high criticality.

**Conducting a stakeholder workshop to capture the "as-is" process:** Most of the time, there is already an informal process in place; it just may not be consistently understood and applied. Getting the process stakeholders together is an excellent way to clarify their understanding of the process, further crystallize its definition, and provide a basis for its execution and improvement. The Process Change Methodology suggests getting stakeholders to identify a list of the key activities that make up the process (including any conditional or iterative sequencing) and then answering the following questions for each activity:

- What is the purpose of this activity?
- Who participates in this activity?
- What inputs are necessary to perform this activity?
- What work products result from this activity?
- How do you know when this activity should begin?
- How do you know when this activity has been completed successfully?
- What three-six subactivities do you perform to accomplish this activity (if the activity is decomposable)?
- How do you determine or measure the performance of this activity?
- What activities are performed before and after this activity?

Building on an existing software process improvement effort: Jones, Northrop, and Soule compared the CMMI models to determine the differences between CMMI process areas and to figure out what is needed to add product-line-specific aspects to those areas [Jones 2002a, Jones 2005a]. Jones further describes how to use an existing software process improvement infrastructure to support product line practice [Jones 2004a].

**Use of Six Sigma techniques to select process improvements:** Continuous process improvement is at the heart of the Six Sigma approach. It provides a philosophy, metrics, an improvement framework, and a toolkit of analytic methods aimed at improving customer satisfaction by eliminating defects. The approach has been used successfully either as the driving framework or a useful adjunct for other process improvement methodologies [Six Sigma 2007a].

**Use of the *Process* pattern:** This pattern specifies the practice areas that require processes and their relationships to the “Process Discipline” practice area. [Clements 2002c, Ch. 7].

**Use of method engineering:** Method engineering involves developing methods and has emerged in response to an increasing feeling that methods are not well suited to the needs of their users, the product developers [Rolland 1997a]. Method engineering practices, which involve integrating theory, guidelines, and tool support, can be used to produce the production method. Method engineering is similar to process engineering, which is why this practice is situated in the “Process Discipline” practice area [Marttiin 1998a].

### Practice Risks

Poor process discipline will result in processes that are inappropriate, vague, unclear, overconstraining, or ineffective. As a result, employees will either choose not to follow them and carry out their work in an ad hoc manner or follow them but fail to achieve the desired results. In either case, the result will be resentment and mistrust of the processes and a poor-quality product line. Potential causes of process difficulties are as follows:

- **process mismatch:** There is a possibility of a process mismatch among a number of factors, such as the organizational structure, the organizational culture, the process that is employed, employees’ experience and expertise, and the market to which the process is applicable. For example, the defined process may be too complex for the organization and therefore result in detailed practices that are not followed. On the other hand, the process may be too simplistic and thus too general and at too high a level to provide practical guidance.

- **process doesn't address the product line's needs:** The process may not accommodate the connection between the processes for core asset and product development. This cross-flow is necessary for product line success.
- **inadequate process support:** Processes must be supported within an organization, especially newly introduced ones. Lack of support (such as training, motivation, templates, and examples) will lead to failure of the process implementation.
- **uneven process quality:** The divergent goals, skills, and backgrounds of development staff may lead to uneven quality in their process contributions. Those divergences may also cause a lack of harmonization in the processes for the different development teams.
- **lack of buy-in:** The organization may not buy into process discipline and therefore fail to adopt it.
- **dictatorial introduction:** One organizational unit mandating the processes that another must follow is likely to result in resentment and the failure to adopt them.
- **processes not enforced:** Even the best processes will fail to have their intended effect if they are not followed. Management needs to enforce the use of defined processes.
- **stagnant processes:** Processes will need to be changed as the organization proceeds along its product line journey. Continuous process improvement is part of process discipline.

### Further Reading

[Boehm 2004b]
Boehm and Turner compare various software development methodologies and provide a risk-driven approach for choosing the appropriate balance between agile and plan-driven processes.

[Curtis 1992a]
This paper written by Curtis, Kellner, and Over is probably the most referenced one in the software process modeling community. It is a must for anyone who wants to understand software development processes and process modeling.

[Humphrey 1992a]
Humphrey and Feiler provide an excellent source of definitions in process terminology that helped to create the basis for the process community.

[Finkelstein 1994a]
Finkelstein, Kramer, and Nuseibeh provide a fairly comprehensive overview of process-centered environments. Their work shows the different approaches and techniques and suggests tools for enacting defined processes.

[Jones 2002a]
Jones and Soule provide a comparison of CMMI and this framework.

[Jones 2005a]
Jones and Northrop provide a description of product line adoption that builds on an existing CMMI approach.

## Scoping

Scoping is an activity that bounds a system or set of systems by defining those behaviors or aspects that are "in" and those behaviors or aspects that are "out." All system development involves scoping; there is no system for which everything is "in." The Rational Unified Process (RUP) includes an inception phase to establish "the project's software scope and boundary conditions, including an operational concept, acceptance criteria, and descriptions of what is and is not intended to be in the product" [Kruchten 1998a]. Kruchten defines scoping as "capturing the context and the most important requirements and constraints so that you can derive acceptance criteria for the end product."

In conventional system development, scoping is usually done informally, perhaps as an unofficial prelude to the requirements engineering activity. In systems designed with ease of change in mind, the activity of scoping helps determine the set of modifications the design will accommodate and those it will not.

### Aspects Peculiar to Product Lines

In product line development, scoping is a fundamental activity that will determine the long-term viability of the product line. Like scoping in general, product line scoping determines what's "in" and what's "out"—in this case, of the software product line. The result is a scope definition, which is a primary output of the Core Asset Development activity and becomes a product line core asset itself. The scope definition identifies those entities with which products in the product line will interact (that is, the product line context), and it also establishes the commonality and sets limits on the variability of the products in the product line.

The scope definition usually begins as a broad, general document that is refined as more knowledge is brought to the table and more analysis is performed. For example, for a product line of World Wide Web software, we would start by declaring that browsers would definitely be "in." Aircraft flight simulators would definitely be "out," and email handlers would . . . well, we wouldn't be sure until the scope was refined further. Hence, the product line scope may not come into sharp focus all at once and that's fine.

The goal of the scope definition is to draw the boundary between "in" and "out" in such a way that the product line satisfies it business and market goals. If the scope is too large, the core assets will have to accommodate so much variation that they will be too complex to be useful and cost-effective in any product. If the scope is too small, the product line may not have enough customers to recoup the investment in the core assets. And if the scope bounds the wrong products, the product line will not find a market. Getting the scope right is important.

The following things drive the scope definition:

- the prevailing or predicted market drivers, obtained through "Market Analysis" practices

- the nature of competing efforts, obtained through “Market Analysis” or “Understanding Relevant Domains” practices
- the business goals that led to embarking on a product line approach, obtained through “Building a Business Case” practices. An example of a scope-setting goal is the merging of a set of similar, but currently independent, product development projects in order to reduce costs.
- technology forecasts that identify expected future technologies, both for the product domain and for software development, obtained through “Technology Forecasting” practices

Scoping identifies the commonality that members of the product line share and the ways in which they vary. Identifying commonalities and variabilities is a theme that pervades virtually every product line practice area; it is the essence of the product line concept and the essence of scoping. Because the scope describes the characteristics of a class of systems, and not specific systems, the scope will apply equally well to existing products and products that have not yet been built or defined completely. The descriptions of scope are essential for determining whether a planned system can be built within the product line and from product line core assets. In most cases, scoping activities must continue after the initial scope has been defined because new market opportunities may arise and new opportunities for the strategic reuse and merging of projects may be revealed.

Scoping may occur in a variety of contexts other than a start-from-scratch product line. For example, an organization may be building (or commissioning) several systems that are similar but not taking advantage of that similarity. The organization may wish to merge those efforts to gain economies of scope. In this case, the initial scope is defined by the list of products that are planned currently. In another case, the organization may aim to capture or penetrate a market segment by establishing a flexible, quick-response capability for launching new products in that market area. In this case, the set of systems may be defined on the basis of marketing projections obtained through “Market Analysis” practices.

In short, scoping answers the question “What products should be in my software product line?” This question is asked at the product line launch and as new product opportunities arise for an up-and-running product line. The candidate product may be a new one brought to the table by a customer or your marketers, or it may be an already existing system being maintained separately by your organization. In the latter case, you might be hoping to achieve economies by merging that system with your product line. By matching a candidate product against the scope definition, you can decide whether that product is “in” or “out” of your product line. If it is “out,” you can judge by how much. At this point, you can decide (via business case practices) whether to expand the product line’s scope to include the new product, develop the new product as a stand-alone system, or turn down the opportunity entirely.

The scope delivers a technical decision that business case factors might override. For example, you may agree to develop a product that is nominally outside the product line’s scope if the customer who wants it is a very important one or it represents an entrée into a desirable new business area. Or, you may decline to build a product that is in the scope if the market for it is small or its opportunity cost is high.

Scoping, as described up to this point, is a way to help inform decision making: when a product opportunity arises, it helps an organization decide whether to bring that product into its product line family. Mature product line organizations often use their scopes to create new product opportunities for themselves. A scope defines an organization's product line area of expertise—the set of systems that it can build efficiently. Thus, scoping can let an organization take the initiative, by providing a basis for discovering products that may have an untapped market. These new products might be squarely within the defined scope, or they might be outside but "nearby." Clements and Northrop chronicle three other case studies that contain different flavors of product line companies that intentionally expanded their scope into new market areas, with excellent results [Clements 2002c]. For example, Cummins, Inc. expanded its scope of automotive diesel engine software to include industrial diesel engine software and quickly entered and dominated an underexplored market.

### Application to Core Asset Development

The scope definition is a core asset for the product line, one that will be consulted extensively and revised as necessary as the product line grows and evolves. The scope definition informs the requirements engineering process, so the requirements-related core assets must be consistent with the scope. We have already shown that market analysis (another core asset) can influence the scope definition. The scope definition can influence the market analysis by identifying places where a new product variant can be produced very efficiently. The market for such a variant might not justify its construction from scratch; however, the market may be robust enough for a product line member to fill the niche nicely.

### Application to Product Development

The scope definition is used during product development to gauge whether a product (that's in your legacy base, in development, or merely being considered) would make a viable member of the product line. That is, the scope lets you decide whether it would be economically advantageous to develop that product using the product line's core assets. Sometimes a product will be clearly in scope, and sometimes it will be clearly out of scope. The interesting situation, of course, is when the product is on the cusp. In that case, a revised market analysis may help to determine whether the organization should produce the product, and then the scope can be adjusted appropriately. If many on-the-cusp products crop up, it may be an indication that the scope should be expanded slightly to include them, assuming that the concomitant expense of fortifying the core assets to accommodate them is deemed (via a business case) to be economically sound.

### Specific Practices

**Applying the *What to Build* pattern:** Applying the What to Build pattern [Clements 2002c, Section 7.6] is an effective way to establish and understand the scope of a product line. The pattern situates the "Scoping" practice area in the broader context of its interactions with other closely related practice areas: "Understanding Relevant Domains," "Building a Business Case," "Market Analysis," and "Technology Forecasting." Information from these practice areas contributes to the definition and evolution of the product line scope by refining the envisioned product set in light of business, marketing, and feasibility considerations. The pattern is applied by iterating through its practice areas until the desired understanding of the scope is achieved. You can apply the pattern quickly to a candidate list of product lines to select the one(s) with the most promise or apply it more deliberately to a selected candidate to probe the soundness of the choice.

**Examining existing products:** Conducting a thorough study of existing products helps identify commonality across a potential product line and identifies the types of differences that are likely to occur. A survey of each group that is developing these products will likely identify future plans, market strategies, and context. In many cases, existing products will contain potential product line core assets that can be mined and used in the future. The steps in this process include

1. Identify existing products similar to those that will be part of the product line.
2. Gather any available documentation and conduct product demonstrations.
3. Conduct oral or written surveys of product experts and the current product developers, users, and maintainers.
4. Identify the products' capabilities, structure, and evolution and any other relevant factors about them.
5. Determine which elements of these products should be considered part of the product line.

**Conducting a workshop to understand product line goals and products:** It is important to gather the potential product line stakeholders together in order to set the direction for the product line. The stakeholders include management, marketing, developers, users, testers, tool developers, technology researchers, and domain experts. Like the market analysis and business case, scoping explores the goals of the product line; the difference with the scoping process is that it examines the product line more from the user's perspective than the organization's. The workshop should work to identify the following:

- the business goals to be satisfied by the product line
- the mapping of product line business goals to the organization's business goals and to users' needs
- descriptions of the current and potential future products that will constitute the product line
- the product and production constraints, which may include platforms, standards, protocols, and processes

The workshop should also establish a coarse-grained schedule that aligns product line development with marketing or overall business strategies.

**Context diagramming:** A context diagram places the product line in the context of product users and other systems and depicts the important entities that affect the product line or are affected by the product line (for example, people, the physical environment, and other systems). The diagram is a generalization across the product line; not every system in the product may connect with all systems or types of users shown in the diagram. Similarly, the context diagram may not show all the interactions of all the potential systems. And even if some product line systems have interactions that are unique to one system, they should still be included in the product line. In addition to highlighting the common context, the context diagram and accompanying documentation should describe possible options and variations. A rationale for the selection of options or variants should also be included. Figure 9 illustrates a context diagram for the software in a personal sound system.

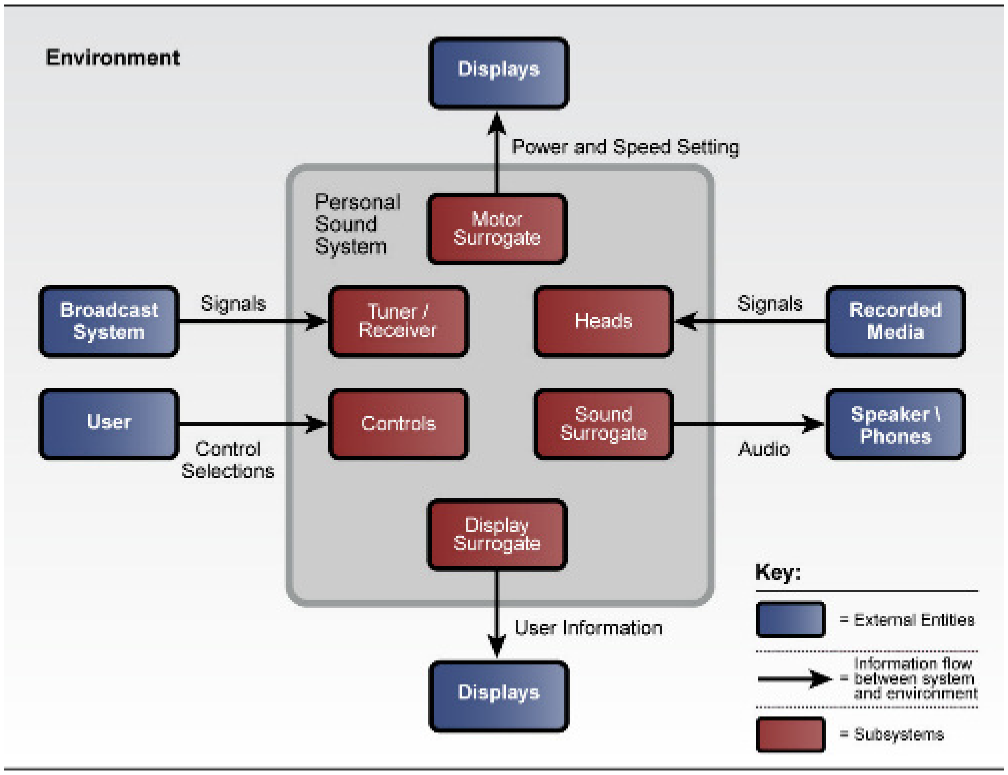


*Figure 9: Context Diagram of a Personal Sound System*

**Developing an attribute/product matrix:** An attribute/product matrix sorts, in order of priority, the important attributes by which products in the product line differ. Typically, the attributes that drive the market are listed vertically down the left side of the matrix, and the different products are listed horizontally across the top of the matrix. For example, in Table 6, attributes include Radio Tuner, Displays, and Audio Control; products include Low-Cost Model, Mid-Priced Model, and High-End Model. The value for the attribute of each product (analog, for example) is listed where the attribute column (radio tuner) and product row (low-cost model) intersect.

*Table 6: Attribute/Product Matrix for Personal Sound System*

| | **Low-Cost Model** | **Mid-Priced Model** | **High-End Model** |
|---|---|---|---|
| **Radio Tuner** | analog | digital presets | digital presets |
| **Displays** | none | frequency | frequency, graphical equalizer |
| **Audio Control** | volume | bass-added | full-spectrum equalizer |

In practice, of course, software attributes are often less tangible. The matrix is used in scoping to define the variability of the product line. By sorting in order of attribute priority, the cluster of the most important attributes that are common across the product can be identified readily.

**Developing product line scenarios:** Product line scenarios are key to defining a product line's scope. They describe user or system interactions with products in the product line. They identify interactions that are common to all products in the product line, as well as those that are unique to a subset of products in the product line. The purpose is to test the context for the scope of the product line. Do entities exist that affect the systems but are not included in the context diagram? If so, must new domains be identified? Along with creating the scenarios, it is important to conduct scenario walkthroughs. These walkthroughs help us understand the support that will be needed to realize the scenarios in product line products.

**PuLSE-Eco:** A method specifically for determining the scope of a product line is called PuLSE-Eco [DeBaud 1999a, Schmid 2001a, John 2006a]. First, product candidates are mapped out, based on input about the system domain and stakeholders. Candidates include existing, planned, and potential systems. The result is a list of potential characteristics for products in the product line. Products and characteristics are combined into a product map—a kind of product/attribute matrix as described above. In parallel, evaluation functions are created, using stakeholder and business goals as input. These evaluation functions will enable you to predict the costs and benefits of imbuing a particular product with a particular characteristic (such as a feature). Next, potential products are characterized, using product maps and the evaluation functions. Finally, benefit analysis gathers the characteristic and evaluation information and determines the scope of the product line.

## Practice Risks

The major risk associated with product line scoping is that the wrong set of products will be targeted. In particular

- **scope too big or too small:** For a product line to be successful, its scope must be defined carefully. If it encompasses product members that vary too widely, the core assets will be strained beyond their ability to accommodate the variability; economies of production will be lost; and the product line will collapse into an old-style, one-at-a-time product development effort. The cost of building very generic components must be weighed against the likelihood that enough product opportunities will arise to make the expenditure worthwhile.

  On the other hand, if the scope is defined too narrowly, the core assets might not be built in a generic enough fashion to accommodate future growth. New product opportunities will either be rejected as being out of scope or be accepted but then result in too much rework. The product line will stagnate.
- **scope includes the wrong products:** Commonalities and variations across current and future systems will include functional requirements, user concerns, and interactions with external systems. In addition, these commonalities and variations will include system qualities, performance issues, and technology evolution. The scoping effort must define a scope that reflects both what is desired for the product line from a marketing perspective and what is feasible from a design perspective. The focus must consider issues that span both the problem and solution spaces. The

risk is that, short of this, the scope will either fail to achieve the goals for product line development or be impossible to achieve given the design and technology constraints.

- **essential stakeholders don’t participate:** The specific practices in scoping require participation from a wide range of stakeholders including management, developers, customers, users, methodologists, and subject-matter experts. The risk is that without sufficient stakeholder participation in planning the product line scope, the buy-in necessary to achieve the desired downstream results (the product line architecture and other core assets) will not exist. Stakeholders’ input is required to reduce the obvious risk that the scope is inappropriate for the product line. More importantly, the involvement of a wide range of stakeholders increases the awareness of product line efforts, obtains stakeholders’ critical input, and builds momentum for the long-term investment in core asset development and use.

## Further Reading

[Cohen 1998a]
Cohen and Northrop discuss product line scoping in the context of object technology. The use of object-oriented approaches such as scenarios, use cases, and frameworks can contribute to our understanding of a product line and the development of a scope for that product line. Their paper also offers an example product line scope, context diagram, and use cases.

[DeBaud 1999a]
DeBaud and Schmid describe PuLSE-Eco—perhaps the most comprehensively documented method for scoping a software product line.

[Fritsch 2004a]
Fritsch and Hahn describe Product Line Potential Analysis, a workshop-based technique developed by Robert Bosch GmbH to help organizations quickly decide whether a product line approach should be adopted for a given set of products and their related target market.

[Jandourek 1996a]
Jandourek describes an approach for platform (or product line) development that was used at Hewlett-Packard. The first step in the approach is product portfolio planning to establish a product lineage chart that defines the organization’s product needs and scopes the platform in terms of the products it must support.

[Robertson 1998a]
Robertson discusses platform planning as a method of achieving software product lines. The planning process uses concepts that are used in product line scoping: (1) identifying the concepts, variants, and options to be embodied in products, (2) determining elements in the products that are common and unique, and (3) listing attributes that will differentiate products.

[Schmid 2000a]
Schmid provides an introduction to the problem of scoping a software product line and describes the close connection between scoping and domain engineering. He further categorizes the then-existing approaches to scoping and discusses the optimization of scoping as it’s applied to product portfolio, domain, and core asset scoping.

[Withey 1996a]
Withey’s investment analysis technique assesses the worthiness of investing in software core assets. An initial step is scoping the product line that will use the assets. The author suggests methods for developing and analyzing the financial effectiveness of scoping decisions.

## Technical Planning

Planning is one of the fundamental functions of management at any level. It provides the basis for the other management functions, particularly tracking and controlling. This practice area is concerned with the planning of projects. By *project*, we mean an undertaking typically requiring concerted effort that is focused on developing or maintaining a specific product or products. Typically, a project has its own funding, accounting, and delivery schedule. In a product line context, a project might be responsible for developing specific core assets or for developing a specific product from the core assets. A companion practice area is “Organizational Planning,” which focuses on the strategic planning that transcends projects.

It is useful to distinguish between the process by which plans are created (the planning process) and the product of that process (the plans). Most planning processes are very similar regardless of the organizational level at which the plan is applied. They usually differ in the personnel involved and the scope of the planned effort. Generally, all planning processes should include

- establishing the plan and its contents
- establishing estimates of the resources required to carry out the plan
- having those who will be bound by the plan review it for feasibility
- establishing commitments to the plan

The planning process needs to be iterative and ongoing—after all, plans change. Plans should be updated and revised as needed during their lifespan.

Different types of plans address different purposes. Examples of project-oriented technical management plans include project plans, software development plans, quality assurance plans, configuration management plans, test plans, and technical risk management plans.

Although the contents of each plan should be tailored to fit its particular use, plans typically contain the following:

- **goals:** A goal is a statement of a desired state that will be achieved by the successful execution of the plan.
- **strategies:** A strategy is a description of a way to achieve plan goals.
- **objectives:** An objective describes a significant, measurable, time-related intermediate state that will be achieved as the plan is executed.
- **a set of activities to perform:** An activity is an assignable, discrete step that helps achieve the specified objectives.
- **resources allocated:** The plan should include an assessment of the resources that the planned activities are allowed to consume (chief among which is time).

Other potential plan contents include responsibilities and commitments, work breakdown structures, resource and schedule estimates, risks, progress measures, relationships, and traceability to other plans.

The most usable plans have a particular focus. Planning a complex task often requires a set of interrelated plans that might have these relationships:

- **temporal relationships:** Some plans might cover a time period that precedes or follows that of other plans.
- **hierarchical relationships:** Some plans contain subordinate details.
- **relationships involving critical dependencies:** Some plans depend on the execution of other plans.
- **relationships based on a supporting infrastructure:** Some plans depend on the existence of an organizational function—for example, a quality assurance or process group.

## Aspects Peculiar to Product Lines

There is nothing fundamentally different about a planning process for a product line. However, certain types of technical plans are unique to product lines:

- **core asset development and maintenance work plans:** These plans describe not only how core assets will be created initially but also how the core asset base will grow and evolve.
- **production plans:** These plans describe how products will be developed from a set of core assets (as described in "Core Asset Development") and include the process to be followed, as well as project details such as the bill of materials and schedule estimates. A *product production plan* lays out the production process for an individual product; the product line production plan is a core asset that applies to the entire product line and from which individual product production plans are derived. In practice, a good deal of the production plan and product production plans are common. Project details such as the bill of materials and schedules are product specific and therefore differ.

Project plans for product lines have a richer set of dependencies than those for single systems. They also have external dependencies among different groups and other plans, possibly at the organizational level, with relation to the core assets. Examples of project plans that may have dependencies include configuration management plans, funding plans, software integration plans, testing plans, and risk management plans.

## Application to Core Asset Development

A product line effort requires work plans for core asset development and maintenance. These plans correspond to standard project plans, except that the "projects" in this case cover the development and maintenance of core assets (or subsets of the core assets). In addition to the generic plan contents identified earlier, these plans should also answer the following questions:

- What is the set of core assets?
- How will core assets be created initially?

- How will core assets be tested (or, in the case of the architecture, evaluated)?
- How will the core asset base be expanded and maintained, and how will components be certified for incorporation?
- How will core asset development and maintenance be funded? The answer to this question should specify the identification of funding sources and a funding profile over time that addresses initial creation as well as sustainment. Because potential funders and stakeholders cut across one or more organizations, planning for core asset funding is likely to occur at the organizational level. In this case, project and organizational planning work in close concert. Project planning might determine funding needs. How to satisfy these needs could be planned at the organizational level.
- How will the configuration of and changes to the core assets be controlled?
- What tools will support the core asset project?
- What measurements will be used to monitor the core asset development and core asset health?

Whether you opt for a separate plan for each core asset or an overall "master" plan for the entire core asset effort depends on your context and culture. Don't overlook the fact that the plans themselves (or parts of the plans) make reusable core assets. Ideally, reusable plans should be tailorable in the same fashion as other core assets—that is, they have defined points of commonality and variability. Cost, effort, and schedule estimates may be particularly useful candidates for reuse, as are work breakdown structures, goals, strategies, and objectives.

In addition to the work plans for core assets, there is the production plan, which is, itself, a primary core asset. To develop a production plan, you need to understand who will be building the products—that is, the plan's audience. Production planning involves a combination of process definition and technical planning activities. The objective is to formulate a production strategy, production method, production process, and production project details that satisfy the organization's goals for the software product line.

Production plans can range from a detailed process model to a much more informal guidebook. The degree of specificity required in the production plan depends on the background of the intended product builders, the structure of the organization, the culture of the organization, and the concept of operations (CONOPS) for the product line. It helps to have at least a preliminary definition of the product line organization before developing the production plan.

The production plan should describe how specific tools are to be applied in order to use, tailor, and evolve the core assets. The production plan should also incorporate any metrics defined to measure organizational improvement as a result of the product line (or other process improvement) practices and the plan for collecting the data to feed those metrics. The production plan, like other core assets, should have variation points that permit tailoring to accommodate the needs of each product.

### Application to Product Development

Each product development effort needs a plan to describe how the specific product will be produced using the core assets. That plan—the product production plan—is an instantiation of the production plan for the product line based on its variation points. The product production plan spells out the production process by including the attached processes for each core asset in that product. It helps the product developers use the core assets effectively to develop products.

- A product production plan should answer the following questions:
- What processes must be followed in order to use and adapt the core assets for use in each product?
- How will any necessary tailoring of the core assets be accomplished? That is, how are the built-in variation mechanisms to be exercised?
- How will any product-unique development be accomplished to supplement the core assets?
- How will products be tested?
- How may these plans interact with core asset development/maintenance plans? Production plans and core asset development and maintenance plans might have the following types of mutual dependencies:
  - What core assets must be available to support particular product development needs? When must they be available?
  - Are any product development projects providing assets to the core asset base? When are those assets needed? What must product development teams do in order to add assets to the core asset base?

In practice, a product production plan might only be derived conceptually—that is, the production process might be so easy or general that you don't have to literally make a concrete instance of it for each product. You can "mentally" derive it. Or it might be implicit due to automated derivation. That said, there are some things specific to each product that will be derived from the project details part of the production plan, such as the bill of materials, schedule, and so forth.

## Example Practices

**Production Planning Workshop:** Chastek and McGregor developed a workshop for developing a product line production plan. It includes a guided session to determine the production goals, breakout discussions to define key development scenarios, a group work session to formulate the production strategy and prioritize the development scenarios, breakout groups to test the formulated strategy, development of a production method, and a discussion of the implications of that method for core asset design. For more information on production plans, see the work of these authors [Chastek 2004a, Chastek 2002b, Chastek 2002c].

## Practice Risks

Poor-quality technical plans can fail to plot the needed tasks and fail to provide adequate resources for product line efforts. The result is a lack of confidence in the planners, missed deadlines, and poor quality due to desperate shortcuts. In particular, plans can suffer from the following:

- **a lack of product line support:** If the additional coordination and commitments required for product line planning are not accomplished effectively, the result will likely be a poorly coordinated effort in which product line benefits are lost.

- **shelfware plans:** If plans are not updated as changes occur, the plans become useless and the process becomes unmanageable.
- **inappropriate plans:** If plans are too aggressive, too detailed, or too abstract, they will not be followed. They must be realistic to the organization and the goals, provide sufficient direction, and offer some flexibility. Some characteristics of a good planning process based on improvement planning are described in the "Organizational Planning" practice area. These characteristics are equally applicable to technical planning.
- **lack of stakeholder involvement:** Product line plans typically have more stakeholders than typical project plans. Unless all the appropriate stakeholders (or at least a representative set of them) are involved in the creation of product line plans, the plans may not be viable for all concerned.

### Further Reading

[Chastek 2002b] & [Chastek 2002c] & [Chastek 2004a]
The works of Chastek, Donohoe, and McGregor provide a comprehensive guide and examples for developing a production plan.

[Diaz 2005a]
Diaz, Trujillo, and Anfurrutia describe how the production strategy accounts for variations at both the product and process level, thus impacting the production plan.

## Technical Risk Management

Risk management is the practice of managing risks within a project, an organization, or a team of organizations. A complete risk management program should provide processes, methods, tools, and an infrastructure of resources and organizational responsibilities to identify and assess the risks (what could go wrong), determine what to do about them, and implement actions to deal with them. We distinguish between the practice areas of *technical* risk management and *organizational* risk management based on the scope and extent of the risks they address and the people likely to carry them out. Like other technical management practice areas, technical risk management addresses project-level concerns.

A *risk* is defined as the possibility of suffering a loss. Therefore, a risk has an associated probability that an event will happen and an associated negative impact or consequence if the risk is realized. Once a risk is realized, it is no longer a risk: it is a problem. A problem is a certainty rather than a possibility.

Having a standard way of stating and communicating a risk provides clarity and consistency. It's a critical component of risk management and serves as a basis for future risk mitigation planning. Risk statements have two parts: a condition and a consequence (see Figure 10). The condition should be based on fact and provides the reader with the anomalous condition or circumstance that causes concern. At least one possible consequence should be noted so that future readers of the risk statement will better understand the original concern of the author.

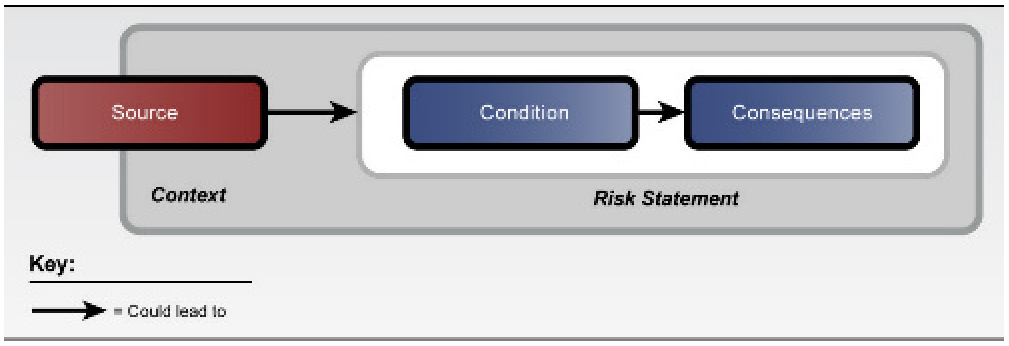


*Figure 10: The Risk Statement with Context*

For example, consider the following risk statement:

> *The commercial off-the-shelf (COTS) high-speed data link selected by the project team was never envisioned by the vendor to be used in a hardened environment; it may not perform as needed, causing rework and integration slips.*

The condition causing concern is the project team's decision to select a component unproven in the product's target environment. Possible consequences are rework and schedule slips. Risk statements that are fact based and actionable allow the project team to start reasoning about the risk constructively. To help in the risk mitigation process, the context under which a risk statement was generated is typically added to the statement. The context is simply additional information regarding the circumstances, events, and interrelationships within the organization that may affect the risk. The context description captures more detail than the basic risk statement; the two of them together help identify the true source of the risk.

### Aspects Peculiar to Product Lines

In product lines, risks most often involve more than one product, so the overall consequences are more far-reaching. While standard risk management paradigms can be applied to product lines, the challenge lies in tailoring the program to the organization and to product line issues. A risk management program is implemented most effectively as an integrated part of organizational or project management rather than as a separate activity.

In a product line approach, some sources of risk may be different or have different emphases, which could affect the process of risk identification. If you use a risk method that comes with its own taxonomy of risk areas or directed questioning techniques, be prepared to modify it in order to stimulate thought about the more common product line risks, such as those identified under the "Practice Risks" heading for each practice area.

Finally, as with all projects, the timing of risk management activities is critical. Since many important decisions are made early in the life cycle of a product line, risk management should be performed when the product line effort is launched.

### Application to Core Asset Development

Applying risk management to a product line approach is fundamentally the same regardless of whether you are developing core assets or products from core assets. Risks identified during core asset development should be managed with particular care, since problems there have a ripple effect throughout the product line organization.

### Application to Product Development

The programs for managing product development risks and core asset risks must be coordinated. Risks identified during product development could involve mitigation strategies that involve core asset development.

### Example Practices

Continuous Risk Management: The Continuous Risk Management (CRM) paradigm developed by the SEI is illustrated in Figure 11 [Dorofee 1996a].

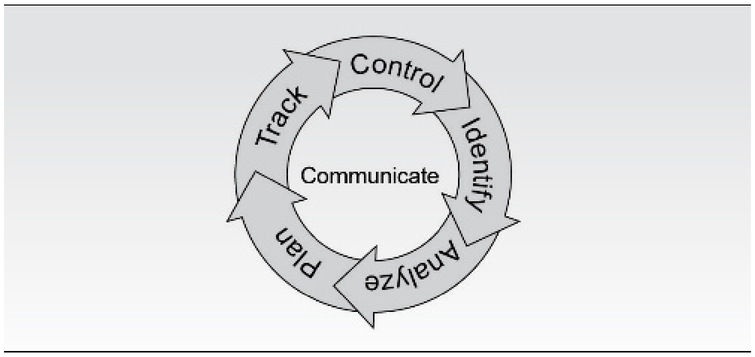


*Figure 11: Continuous Risk Management (CRM) Paradigm*

In this paradigm, risks are first *identified* and then *analyzed* to determine their probability of occurrence and impact (risk exposure) on the organization, the interrelationships between individual risk statements, and which related risks (risk areas) are most important to mitigate. Mitigation plans are developed for the most important risk areas. Individual risks, mitigation plans, and the risk process are all *tracked* to determine the effectiveness of mitigation actions and the risk process. *Control* decisions (for example, close, replan, continue tracking) are made and documented. The diagram is circular to emphasize that effective risk management should be a continuous process. This is in contrast to a risk management program that might identify risks only once at the start of a project and never revisit them.

*Communication* is represented as an encompassing activity to emphasize that the flow of information throughout the project or organization is essential to successful risk management.

**Installing a Risk Management Program:** An approach to installing a risk management program consists of the following steps adapted from the approach described by Dorofee and colleagues [Dorofee 1996a]:

1. Build the initial awareness and infrastructure.
2. Establish a risk baseline and mitigation plans.
3. Develop a risk management plan that is adapted to existing structures and practices.
4. Install risk management in a pilot project.
5. Improve and expand the risk management implementation.

These steps, which view risk management from somewhat of a project focus, are described below. As mentioned previously, a product line approach requires an intimate association between core asset developers and product developers. Thus, a more comprehensive approach to risk management—such as team risk management—that crosses organizational boundaries may be desirable. For more information on that approach, see the “Organizational Risk Management” practice area.

**Step 1. Build the initial awareness and infrastructure:** In this step, management is made aware of risk management benefits and implementation requirements. A management commitment is required before the project can proceed. Initial awareness training, including motivation, is conducted within the organization, and an implementation team is established.

**Step 2. Establish a risk baseline and mitigation plans:** This step involves examining risks in a comprehensive fashion and creating an initial set of risk mitigation plans. Dorofee and colleagues describe one technique for establishing a risk baseline [Dorofee 1996a]. A key element of this step is risk identification—something that can be done using the SEI Risk Taxonomy-Based Questionnaire (TBQ) supplemented with risks from the practice areas in this framework. Practitioners have used the TBQ successfully to consider possible sources of risk [Williams 1999a]. Other sources of risk are described by Boehm [Boehm 1981a] and Fairley [Fairley 1994a]. After establishing a basic set of risks, you should analyze them in terms of probability, impact, and time frame; this analysis will help you prioritize them. Mitigation plans should be created for the highest priority risks, and the person or team responsible for executing those plans should be identified.

**Step 3. Develop a risk management plan that is adapted to existing structures and practices:** A risk management plan should detail the processes and organization that will address risks. As mentioned previously, CRM should be integrated into existing practices, so the first step is to analyze the existing structures, functions, and processes by asking

- What meetings and reviews are already in place?
- Who is involved?
- What tracking and control mechanisms are already in place?

This analysis can help you discover opportunities for inserting the basic risk management practices (that is, identify, analyze, plan, track, and control) or uncover the need to add additional practices or functions. The risk management plan should address how these functions are to be incorporated into standard operations [Loveland-Link 1999a].

**Step 4. Install risk management in a pilot project:** This is the trial implementation of the risk management plan. The project chosen should be representative of the organization and supported by project management. In addition, it must capture any lessons learned and work collaboratively with the risk implementation team.

**Step 5. Improve and expand the risk management implementation:** In this step, lessons learned from the pilot project are applied to improve the risk management practices. Implementation is then expanded more widely throughout the organization. Depending on the organization, this expanded use may be instituted in phases. The continued improvement of risk management practices should be ongoing.

## Practice Risks

An ineffective risk management program will result in problems that catch management by surprise and can possibly lead to decreased morale, low product quality, missed deadlines, and failure to make progress. Some of the risks associated with the introduction of a risk management program, which include those common to any improvement effort, are as follows [Radice 1994a]:

- insufficient sponsorship by senior management
- resistance by middle managers
- termination of activities before the practice is institutionalized
- a lack of sustained focus on improvement

The biggest risk related to risk management is the failure of the existing organizational culture to encourage and reward risk identification. In such a “risk-averse” organization, people are pressured not to raise risks, either overtly or implicitly. Without broad participation in risk identification and management, a risk management program cannot succeed. Table 7 shows some barriers that exist in a “risk-averse” organization.

*Table 7: Potential Communication Barriers [Dorofee 1996a]*

| **Potential Barrier** | **Description** |
|---|---|
| “ready, fire, aim” approach | People provide solutions to a problem before they have assembled and understood the underlying facts associated with the problem and its context. |
| “don’t tell me your problem” attitude | People require a solution before they even discuss an issue: for example, a manager says, “Don’t bring me problems; bring me solutions.” |
| “shoot the messenger” attitude | A project member who intends to inform others or is looking for help suffers negative consequences because he or she is communicating unpleasant information. |
| “liar’s poker” approach | Project personnel identify risks but fail to communicate them to others. Instead, they wait until the risks become serious problems that impact project schedules and product quality. |
| mistrust | Individuals do not trust each other for a variety of reasons (such as, past history and preconceived biases). This lack of trust can destroy any credibility in the acquired risk data which, by its nature, is subjective and speculative. |
| hidden agendas | Situations create individual preferences for results: individuals or groups may promote facts or arguments based on their own goals rather than the common good. |

| | |
|---|---|
| “placing blame” approach | Risk information is used to place blame on project personnel. |

### Further Reading

[Boehm 1989a]
Boehm wrote an IEEE tutorial on software risk management that provides some valuable insights.

[Charette 1989a]
Charette’s book is the foundational text on software risk management.

[Williams 1999a]
Williams, Pandelios, and Behrens describe the SEI Software Risk Evaluation (SRE) method.

## Tool Support

Software development organizations use tools to support many of the activities necessary to transform a set of customer needs into a useful product. A multitude of computer-aided software engineering (CASE) tools are available to help automate the analysis, design, implementation, and maintenance of software-intensive products. The challenge is to choose and use tools wisely to support the business goals of the organization and the technical needs of the product developers.

Typically, many tools are used over the life of the project. The degree to which they interoperate and support the development process can have a large effect on the productivity of the development team and the quality of the resultant products [Bruckhaus 1996a, Low 1999a]. Standards have emerged to provide guidance in the evaluation, selection, and adoption of CASE tools [IEEE 1998a, IEEE 1996a, ISO 1995a], and several authors have described approaches to the tool integration problem and the results of tool adoption studies [Brown 1994a, Brown 1994b, Bruckhaus 1996a, Budgen 2003a, Favre 2003a, Jansen 2004a, Low 1999a, Maccari 2000a, Powell 1996a, Vollman 1994a].

### Aspects Peculiar to Product Lines

If you’re building a software product line, a variety of tools will likely be used, since there is no single tool that addresses the needs of all product line practices. While many of the practices supported by tools are applicable to product lines, the product line context brings some of its own particular needs and risks of its own.

This practice area does not address the related issue of choosing and using technologies (such as distributed object technology) that may be new to a product line organization and for which tool support may be required. The focus here is on applying familiar tools to practices in a product line context.

The critical aspect of this practice area is support for concurrently creating, maintaining, and using multiple versions of product line artifacts—both core assets and products. That support requires a development environment that facilitates the coordination of core asset development and product development teams and processes and the sharing of core assets among teams.

Tool support for a product line is therefore more than the sum of the capabilities of individual tools that support specific software engineering activities. The interoperability of a chosen set of tools is critical for the automated production of products in a product line from core assets. An integrated product line support environment will

- support the product line development process and the organizational structure that implements it
- enforce the conceptual integrity of the product line based on the definition of what it means to be a member of the product line, including any options for tailoring a product to specific needs without violating the defining properties of the product line. That definition comes from the "Scoping" and "Requirements Engineering" practice areas.
- represent the common and variant capabilities of products in the product line in all places where they are articulated: requirements, architecture, components, testing, plans, and elsewhere
- maintain traceability and dependency links among core assets and products, for product line maintenance and evolution, through a configuration management (CM) capability that covers the entire product line
- support an architecture-based development process with multiple views of the product line architecture that permit reasoning about architectural qualities
- support the "steady state" operation of the product line by enabling the production of specific products from a specification of their desired capabilities
- provide insight into the business and technical decisions for the product line by supporting technical management practices such as scoping and measurement
- evolve with the product line to incorporate new technologies and new production strategies and processes as the organization's needs change

Together, these items represent a vision of an integrated product line tool environment. Although we can expect reality to fall somewhat short of this vision, remembering the ideal helps us nudge the real environment in that direction. Jansen and Bosch, for example, examined the issue of tool support for architectural evolution [Jansen 2004a]. They rated five tools against the requirements needed to support architectural design decisions and, not surprisingly, found plenty of variation in the results. Their conclusion was that architectural tool support does not view evolution as an inherent part and distinct dimension of software architecture.

Establishing tool support for a product line involves the following activities:

- **identification of needs:** When choosing tools to support a product line effort, be clear about which specific needs the tools are to address, including the expectations of the various people who will install, use, and maintain them. Don't overlook the essential need for documentation of the product line—for example, describing the business case for the product line, describing the variation points in the product line architecture, and describing how a product engineer creates products from core assets. The future needs of the product line are also a consideration: investigating how new technologies might support the product line, identifying emerging tools in these new technological areas, and communicating future needs to tool vendors.
- **selection:** Base your tool selection on well-defined criteria rather than on market literature or anecdotal evidence of suitability. The specific practices suggested below offer a variety of criteria.

- **evaluation:** Ideally, when assessing a tool for fitness of purpose, conduct a trial use before making a final commitment. A tool may look promising because of its claimed ability to, say, support architecture variability, but a trial use may show that variability to be incompatible with what the architects need to reason about a product's structural properties. Similarly, interoperability and process-support considerations may nullify any gains made through the satisfaction of particular needs.
- **insertion:** Insertion includes adapting tools to the environment in which they will operate, training tool users and maintainers, and adapting or adopting the development or business processes and practices that are necessary for the effective use of the tools.
- **measurement:** The only way to determine a tool's benefits and limitations is to systematically track the effects of its use on an organization's productivity and product quality. The appropriate data should be collected and compared with prior experiences to determine the tool's value in practice.
- **maintenance:** The environment provided by tools needs to evolve with the product line to incorporate new technologies and new production strategies and processes. When evaluating a tool, be sure to consider whether it can be evolved and whether its vendor is open to making evolutionary changes.

### Application to Core Asset Development

Tool support for core asset development focuses on the commonality and variability of the core assets from which product line instances will be created. An ideal tool set for core asset development supports the development of both core assets and mechanisms for using them to construct products.

Tools are needed for developing all sorts of core assets, including those for constructing documents, code, test support, and installation—in other words, those that support adopted product line practices. For example

- Product line scoping practices are facilitated by tools that capture and represent the common and variant capabilities of products in the product line.
- Requirements engineering practices for product lines necessitate tools for describing the requirements of the products in the product line.
- Architecture definition practices for product lines require tools for describing the structures of products in the product line in a way that is consistent with the identified commonality and variability for the product line.
- Component development practices for product lines require tools for creating and testing components that are consistent with the identified commonality and variability and can be used to construct products.
- Production plan practices require tools for creating mechanisms that support the development activities of product development teams.
- Product production using model-driven approaches such as domain-specific languages and automatic code generation requires tools for constructing the languages, creating the models and product specifications, and generating code from the models.

## Application to Product Development

The exact role of tool support in product development depends on the degree of automation that has been decided on during core asset development and production planning. The development of products may be partially or wholly automated; in the extreme, the products may actually be generated by tools that are created specifically for the product line. In that case, tool support is extensive; the software engineering environment is a production constraint that must be factored into the architecture design and component development for product developers, and it is a significant element of the production plan.

The means of product development are specified in the product line's production plan. Tool support for product development is a particularly important part of that specification because the chosen practices and tools must be compatible. The nature of the production plan can differ substantially depending on the degree to which product development activities are to be automated [Chastek 2002b, Chastek 2002c].

## Example Practices

Current tool support for product lines is based on a variety of tools used in the software engineering activities of conventional development efforts and then "stretched" to accommodate product lines [Bass 2000a]. Using these tools successfully in a product line context requires practices that emphasize the tools' fitness of purpose (both individually and collectively), the quality of the software produced using them, their effect on the product line development process, and the business benefits of using them. If an organization's workforce is geographically distributed, tool selection and enforcement must be an early consideration.

The practices discussed below are all based on established practices for CASE tool support and adapted to the product line context.

**Identification of needs:** In addition to tools that directly support software development, product line tool support is essential for CM, planning, and all types of documentation. All of these activities are good places to begin identifying specific needs.

**Selection:** Bass and colleagues provide a comprehensive checklist of tool selection criteria for software product lines [Bass 2000a]. Example criteria include tool features, cost, vendor stability, training, interoperability with other tools, and process implications. Powell and colleagues describe an evaluation process that generates selection criteria from a checklist of issues [Powell 1996a], and the longest section of the International Organization for Standardization (ISO) standard 14102 [ISO 1995a] deals with the characteristics by which tools can be selected and evaluated. Other practitioners advocate tying the selection and integration of tools to the development process to be used [Brown 1994a].

**Evaluation:** Brown advocates an evaluation approach based on the desired quality attributes of the entire tool support environment, not just the individual tools [Brown 1994b], and Garlan, Allen, and Ockerbloom describe integration woes that have particular relevance for product lines [Garlan 1995a].

The evaluation process described by Powell and colleagues [Powell 1996a] and the Institute of Electrical and Electronics Engineers (IEEE) standard 1348-1995 [IEEE 1996a] include guidelines for the pilot use of a tool. Favre and colleagues observed that even when tools are quite close to solutions and based on well-defined concepts, there are still many important barriers to adoption; those authors list a range of issues from usability and customization to evolution and organizational strategy [Favre 2003a].

**Automation:** Automated derivation of products requires an explicit tool chain that begins with domain concepts and ends with executable code [McGregor 2005b]. There are several paths between these two end points, and the tools are somewhat different along each one. Basically, automated derivation is based on some type of modeling language that is specific enough to have a well-defined semantics yet at a high enough level to allow direct involvement of customers. McGregor lists five main approaches to automatic derivation:

1. Specification-Based
2. Intelligent Build
3. Domain-Specific Languages (DSLs) and Product Generation
4. Metamodeling
5. Frame Technology

These approaches are not at the same level of abstraction, and they are not mutually exclusive. Each one requires tool support. Consider two examples that differ in the grain size of the units of derivation:

- DSLs provide a relatively fine-grained representation from which products can be developed [Czarnecki 2005a]. Using tools that support domain modeling approaches such as feature modeling or the more general conceptual modeling, a model is created using a vocabulary specific to a domain. Domain experts can then specify products in a language they understand. These modeling tools are augmented by a variety of other tools that provide a means for expressing constraints, code generators that map model elements to specific code fragments, model simulators that allow execution of the model, model verifiers, and model-driven testers. All of these tools are needed for a complete development environment, and many of them can be generated automatically from the domain model.

  Batory defines one particular tool chain for product specification that starts with a tool that uses a feature model configuration to specify a product [Batory 2005a]. The model is maintained in a Logic-Truth Maintenance System (LTMS) and uses a propositional satisfiability (SAT) solver to prevent inconsistent specifications. The feature-based specification can be mapped onto a grammar from which various techniques can be used to produce products.
- Frame technologies are an easy way to begin automation in a product line because they tend to be used with large code fragments. These techniques generate products and assume that code fragments have been created so that they can be combined in a variety of ways to make many different products [Zhang 2005a]. The tool support for this approach is based on Extensible Markup Language (XML). The overhead is small because most XML-oriented tools can be used with a small specialized engine.

Both domain-specific languages and frame technologies are based on the use of metamodeling techniques. Metamodels provide layers of abstraction that hide the complex mechanisms of execution behind the familiar concepts of a domain. The metamodels provide a foundation on which tools can be based, allowing new tools to be developed and deployed quickly.

## Practice Risks

Tool support makes some tasks more convenient and others (such as extensive unit testing or CM) practical where they otherwise would not be. Inadequate or inappropriate tool support has a devastating effect on productivity in all parts of a project where tooling could be applied. Specific tool support risks include the following [Bass 2000a]:

- **lack of product-line-appropriate tools:** The lack of true support for product lines often results in frustration at best and longer development cycles at worst. For example, if tools cannot coherently represent information about multiple products (especially their commonalities and variabilities), the entire concept of creating multiple products from the same core asset base is undermined. Users may be forced to extend existing tools until they become a heavy maintenance burden. New releases of a tool may invalidate the previous extensions, thereby risking time-consuming rework that will further lengthen the development cycle.
- **lack of support for variability:** Controlled variability is essential for creating a product line that is flexible enough to accommodate changing customer needs. If your tools cannot help you represent and reason about variability early in the life cycle, you may not be able to
  - express variability in the architecture, particularly in graphical form
  - trace requirements variability to architecture variability
  - trace component variability to architecture variability
  - trace testing variability to component variability
- **lack of appropriate CM tool support:** Product lines require sophisticated CM practices and robust tool support. Insufficient CM support (especially for capturing and correlating data about changes across multiple products) will chip away at an organization’s ability to field products quickly and respond to requests for changes. Releases of the core assets that are not synchronized with product releases are another source of disruption to the overall product development schedule.
- **lack of tools supporting specific practice areas:** If adopted tools fail to support adopted practices, the effort required to perform the latter may be excessive.
- **incompatibility between tools and practices:** If an organization chooses tools and practices separately, their use may conflict, resulting in duplicated effort or failure to use the tools or to carry out the practices properly.
- **incompatibility between core asset development and product development:** If product development uses tools or practices that are different from what core asset development anticipates, the provided core assets may not work well with them.
- **lack of interoperability:** If the selected tools will not interoperate, the result will be inconsistencies and gaps between different product line activities.

- **lack of adoption:** The tools will fail to be adopted by the relevant users. As a result, a heavy investment may be made in tools with little payback.
- **inappropriate use of automated derivation:** If the automated modeling support and/or the domains associated with the product line are not sufficiently mature, the strategy for automated derivation may fail to efficiently yield products within the agreed-upon scope.
- **insufficient attention to detail in model building:** If the organization lacks the discipline to define models completely (to the depth required by the tool chain), the resulting products may not operate as anticipated, if at all.

## Further Reading

Both the IEEE and ISO have issued comprehensive standards for CASE tool selection, evaluation, and adoption.

[IEEE 1996a]
The IEEE standard views CASE tool adoption as more than just the selection of CASE tools; adoption requires the planning and implementation of an entire set of technical, organizational, cultural, and management processes to achieve desired improvements in software development. In addition to the steps for defining CASE needs and evaluating and selecting CASE tools, the practice also includes guidance for conducting a pilot and fostering the routine use of adopted tools. Annexes to the standards provide additional information on aspects of the practice, including defining the: organizational CASE goals, needs, and expectations; approaches to developing an adoption strategy; and evaluation criteria for a pilot.

[ISO 1995a]
The ISO standard provides guidance on identifying organizational requirements for CASE tools and mapping those requirements to the characteristics of candidate tools to be evaluated. It also describes a process for selecting the most appropriate tool from a candidate set based on the measurements of the defined characteristics. An appendix discusses the advantages and disadvantages of three kinds of selection algorithms. (The ISO standard ISO/IEC 14102 has been adopted by the IEEE as IEEE Std.1462-1998 [IEEE 1998a].)

[Bass 2000a]
Bass and colleagues report on what a group of product line practitioners had to say about tool support for product lines.

[Krueger 2002a]
Kruger situates the problem of variation management for a software product line in the context of a multidimensional CM problem and describes a tool called Gears built specifically for software product lines.

[Lundell 2004a]
Lundell and Lings survey the historical roots of the factors contributing to the tension between user expectations of CASE technology and the actual products in the marketplace.

[McGregor 2005b]
McGregor surveys automatic derivation techniques for product lines, including the related technologies, assumptions, and approaches.

[Ossher 2000a]
Ossher and colleagues present a roadmap of software engineering tools and environments based on their personal view of the key issues, themes, and future challenges.

Industry evaluations: Ovum (www.ovum.com) produces periodic, comprehensive evaluations of software engineering technologies. The company evaluated software CM tools in April 2005 and software testing tools in December 2005 and January 2006.

Open source resources: Eclipse (www.eclipse.org) is an open source community whose projects focus on providing an extensible development platform and application frameworks for building software. The Eclipse Platform is designed for building integrated development environments (IDEs); subsets of the platform can also be used to build arbitrary applications. Another open source community effort, NetBeans (www.netbeans.org), provides an IDE and a NetBeans Platform.

## Organizational Management Practice Areas

Organizational management practices are those practices that are necessary for the orchestration of the entire product line effort. They are

- building a business case
- customer interface management
- developing an acquisition strategy
- funding
- launching and institutionalizing
- market analysis
- operations
- organizational planning
- organizational risk management
- structuring the organization
- technology forecasting
- training

### Building a Business Case

A business case is a tool that helps you make business decisions by predicting how they will affect your organization. Initially, the decision will be a go/no-go for pursuing a new business opportunity or approach. After initiation, the business case is reviewed to assess the accuracy of initial estimates and then updated to examine new or alternative angles on the opportunity. As an important communications vehicle, the business case identifies the goals and measures for tracking the move to the new business or approach. It includes the methods and rationale used for quantifying the benefits and costs and lists the critical success factors and contingencies that must be managed in order for the predicted results to appear [Schmidt 2003a]. By documenting the expected costs, benefits, and risks, the business case serves as a repository of the business and marketing data. In this role, management uses the business case to determine possible courses of action.

A business case addresses the following key questions that an organization faces when planning major changes in how it does business:

- What specific changes must occur?
- What are the benefits of making the change?
- What are the costs and risks?
- How do we measure success?

An effective business case must convince management that the investment is financially sound, is realistic for the organization, is aligned with other business strategies, and has a clear course of action for putting the change into effect. Business case results are often summarized using several well-defined financial metrics such as net cash flow, discounted cash flow, internal rate of return, and payback period [Schmidt 2003a]. Business cases may also be made based on opportunities that result from taking a certain course of action, for example, opening up new markets. Net options value is one such approach that may be incorporated into a business case. It has been applied specifically in making decisions regarding software modularity [Baldwin 2002a].

The need for a change is often precipitated by a market analysis that tells an organization what it needs to do in order to stay competitive in a particular mission or market area. That analysis begins the process of defining the change. The business case then determines the best approach for meeting those needs. A business case may consider multiple alternatives or look at one proposed solution. In either case, it compares how business is currently handled to how business will have to be handled in the future if the company pursues a potential business opportunity.

The business case documents how closely aligned the opportunity is with established business goals for such things as

- reduced time to market
- reduced cost
- higher productivity
- improved quality
- increased customer base or bigger market share
- ease of upgrades

It reinforces the motivation for making the change by offering a broad, quantifiable assessment of the opportunity. The goal of a business case is to provide management with a sufficient understanding of the approach and adequate data to determine if the projected return on investment (ROI) is sufficient to justify the proposed venture.

The business case should be maintained as a separate document. There is no standard template for a business case, but it should address the following tasks [Humphrey 2000a]:

1. deciding what to do: list any assumptions (market conditions, organizational goals, and so on), develop alternative approaches, and then either choose one or decide to build a comparison
2. estimating the likely costs and potential risks of all alternatives
3. estimating the likely benefits contrasted with the current business practice
4. developing a proposal for proceeding
5. closing the deal: how to make final adjustments and proceed to execution

### Aspects Peculiar to Product Lines

A business case in a product line context can serve one of two purposes. The first is to justify the effort to adopt the product line approach for building systems. The second is to decide whether or not to include a particular product as a member of a product line.

A software product line effort represents an investment in resources and technology. Any organization that adopts a product line approach should have sound business reasons, backed up by data and experience, for doing so. Specific time-to-market improvement, product quality goals, cost targets for product development and delivery, new market growth, and product risk reduction are factors that are often included in the business case. The business case should identify the customers for the products that will be part of the product line, as well as the costs and benefits to those customers and the organization producing the products. Also, it should be directly supportive of higher organizational and/or corporate goals and vision. The business case should be agreed on, documented, communicated to the entire organization, and then validated by market analysis and organizational experience and expertise. The business case includes the product line goals that will, in turn, drive the data to be collected and the measures to be tracked.

Business case practices for product lines differ only in the nature of the changes being considered and analyzed. The organization is making an economic case built on the current costs of doing business versus a product line approach. Here, the initial go/no-go decision answers the question: “Do we build the set of products we’re considering as a product line or not?” As part of the business case analysis, the organization determines how many products are likely to be built in the product line over a certain time, who the customers will be, and whether a product line approach compares favorably with other business opportunities [Reifer 1997a].

The business case reflects the facts and assumptions from the examination of relevant domains, the product line scope, and the market analysis, and it answers the following ancillary questions:

- Do we have the right capability and resources to launch a product line?
- Can we leverage our domain understanding to provide a unique opportunity and create market demand for our product line?
- What are the financial and business consequences of adopting a product line approach?

The business case may determine that product lines are not a viable approach. For example, if the market for future products is small or won't support more than a very few product variants, there is little incentive to invest in a product line for those products. Predicting future products may be difficult if the market is unstable, so the business case may also propose alternative production methods for a product line approach. For example, investment in software generator technology may be recommended if a large number of very similar products are likely, or in manual software composition if the market forecast is for smaller numbers of products. Identifying alternative product line approaches for the business case helps assure management that all the options have been considered and that a single strategic reuse decision is not being forced on them. The business case may also propose a business model, such as fielding an architecture and components that system developers from other organizations will use for their products. In any of these product line situations, management will still expect the business case to define the change being proposed, how it differs from current practice, why it's better, its financial consequences, and how management will know whether the goals are being met.

Based on the initial scope of the product line, you can estimate and compare the likely costs and potential risks of each alternative approach. Figure 12 is an example of the cumulative cost estimates for three successive projects, both with and without taking a product line approach. The cumulative cost of the three projects *without* taking the product line approach is shown by the columns. The sloping lines show the cost with the product line approach, as follows:

- There is an initial start-up cost (shown in Figure 12 as 30 units of effort) for moving to a product line approach. In addition to costs for developing core assets, the business case must indicate the cost of adopting processes for product lines, including the costs of training, incentives, and tool development or procurement. In the figure, this cost is shown as accruing even before the launch of the first project.
- With each successive project, core assets must be maintained and enhanced, and new core assets added. Thus, the cumulative cost for developing core assets increases over time (as shown by the lower line in Figure 12).
- In Figure 12, the "Production Cost with Assets" line represents the cumulative effort of developing all three projects shown. Project cost includes the start-up cost, the cost of enhancing the core asset base for that project, and the cost of project-specific development.

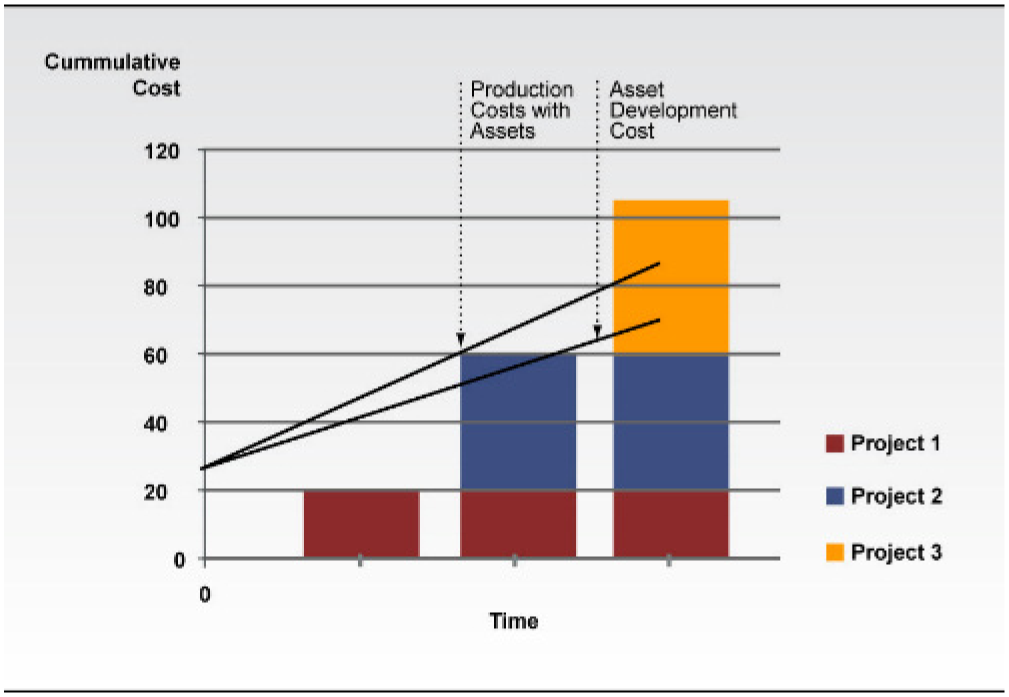


*Figure 12: Cost Basis for Business Case Assessment*[8]

The figure shows that, in this example, the third project represents the payoff point for the product line approach, because the cumulative cost of the third project without the product line approach exceeds the cumulative cost of the third project with the product line approach.

If cost reduction is a key business driver and the cumulative cost of production with core assets is greater than the cost without core assets at some desired break-even point, the organization will likely make a no-go decision. Figure 12 represents only the cumulative production costs, however. If other factors, such as the time to market, market share, or market agility, are drivers and those goals can be met through the product line approach, the organization may forgo an early break-even point on costs in favor of the other factors.

Organizations also estimate anticipated benefits based on market and historical data. These benefits may include factors such as productivity increase, defect reduction, time to market, or reduced costs of integration. The business case may propose alternative approaches and prioritize them on the basis of relative cost versus benefits and risks. In making specific recommendations, the analysis included in the business case looks at the approaches in terms of meeting or exceeding criteria established by the organization.

8 In Figure 12, cost is synonymous with effort.

## Application to Core Asset Development

First of all, the business case for the entire product line is, itself, a valuable core asset that should be documented, maintained, and periodically revisited to make sure that the organization's goals are still being adequately served. Because it will inform the product line's scope (see the "Scoping" practice area), the business case must be available and current. Second, the business case for an individual product can be reused with some variation when the next product decision has to be made.

Beyond being a core asset in its own right, the business case for the product line is used to justify the effort to build other core assets. The development of the initial business case occurs during an early cycle of product line activity, dedicated to making the initial go/no-go decision. If the business case proposals are accepted and product line development gets underway, the business case supports core asset development, as a living document designed to reflect changing market conditions and the coordinated product line response and to measure the achievement of desired results. In this later cycle, the organization may develop a business case to determine whether to extend the product line scope, add new components, pursue new customers, or address other new opportunities related to the product line.

## Application to Product Development

As products in the product line are developed, the role of the business case evolves within the organization to become a more tactical document. It supports the decision about whether to develop a proposed product as a member of the product line. In this case, the same questions apply, suitably modified:

- Do we have the right capability and resources to build this product as a member of our product line?
- Can we leverage our domain understanding to provide a unique opportunity and create market demand for this product?
- What are the financial and business consequences of including this product in our product line?

Strategic considerations may apply. For example, a strong business case may be made for a product that, by itself, will not be profitable but will give the organization a toehold in a new application area.

In addition, the business case supports decisions to direct or redirect resources during the product development and evolution phases for

- further requirements analysis to extend the core asset base
- core asset refinement in response to product development
- new architectural aspects/development
- product development in new areas
- continuing analysis of market conditions

To support the development of cost data, the business case must consider the financial and other business consequences of the chosen production method. There will be several ways for an organization to produce products. Factored into the business case are the cost, benefit, and risk consequences of the selected production method. Another business case decision considers users of core assets who will be producing the products. The organization developing the business case may be funding core assets for internal development or multiple external development organizations. These approaches may also lead to alternative considerations in the business case.

## Example Practices

**"Business case lite:"** Sometimes circumstances make a business case extremely straightforward. In the case of CelsiusTech, the simultaneous awarding of two massive contracts (each of which was for a system beyond anything the company had ever attempted) precipitated a business case roughly as follows: "If we don't build these two systems based on a common set of core assets, our company will cease to exist" [Brownsword 1996a]. The implicit go/no-go decision resulting from this business case does not require much sophisticated analysis to resolve.

The CelsiusTech case is extreme but authentic (and not uncommon). There are less extreme cases that result in simple business cases as well. The FAST product line engineering method [Weiss 1999a] got its start not through the building of entire systems but rather through the building of different versions of highly changeable, relatively small subsystems restricted to a single domain (electronic message switching). The business case for a small number of small products is simplified, because the risk and required early investment are also small. In the case of FAST, the early costs included the overhead of the commonality and variability analysis plus the cost of designing a generator language and producing the corresponding generators—all of which were easily measured in person-days.

**Estimating the likely costs and benefits:** For each alternative, the organization makes reasonable estimates of costs that may be accrued at different times:

- **initial costs:** These occur when the product line's core assets are developed and the initial products are fielded.
- **incremental costs:** These occur whenever the product line is extended with new core assets. The extensions include improvements within the existing scope or an extension of the scope itself.
- **product development costs:** These are associated with using core assets to develop products.
- **annual costs:** These are upgrades made and annual maintenance costs incurred to fix defects. These costs may accrue for products or core assets.

The U.S. Federal Aviation Administration, Reifer, and Schmidt provide a variety of techniques for making cost estimates including [FAA 1995a, Reifer 1997a, Schmidt 2003a]

- best practices
- analogy or historical information
- expert opinion
- prototypes or pilots
- parametric cost estimating, such as the Constructive Cost Model (COCOMO)—a well-known empirical cost estimation model

The organization must use the cost and risks of the current way of doing business to assess the benefits it hopes to realize through use of the product line approach. Benefits should be forecast for the short term as well as the long term, and these factors should be contrasted with "as-is" data.

The developers of the business case should resolve which of these (or other) goals are the primary drivers for deciding whether to launch a product line. Then, they should set ways to measure performance against these goals and identify indicators of success. Armed with the goals to be achieved, measures for tracking goals, and the timetable for achieving goals, you can make a reasonable business case. See the "Measurement and Tracking" practice area for more details.

**Economic modeling:** A business case for a software product line is likely to include an argument that the software product line will bring about economic advantages to the developing organization. An economic model can be used to support that claim. After specific costs and benefits have been identified, they can be estimated and aggregated to form an economic justification of the product line. This model can be used to calculate the potential return on investment from use of the product line strategy or to answer many what-if questions regarding various decisions about the product line.

The Structured Intuitive Model for Product Line Economics (SIMPLE) is an economic models specifically geared for software product lines [Clements 2005a]. It consists of four cost functions and a benefit function.

- $C_{org}()$ is a function that, given the relevant parameters, returns the cost to an organization of adopting the product line approach for its products. Such costs can include reorganization, process improvement, training, and any other necessary organizational remedies.
- $C_{cab}()$ is a function that, given the relevant parameters, returns the development cost to develop a core asset base suited to satisfy a particular scope.
- $C_{unique}()$ is a function that, given the relevant parameters, returns the development cost to develop unique software that is not based on a product line core asset base.
- $C_{reuse}()$ is a function that, given the relevant parameters, returns the development cost to reuse core assets in a core asset base. $C_{reuse}$ includes the cost of locating a core asset, checking it out of the repository, tailoring it for use in the intended application, and performing the extra integration tests associated with reusing core assets.
- $B_{benj}(t)$ is a function that—given a specific benefit, *ben*$_j$, of the product line strategy—returns the value of that benefit. A benefit function is defined for each benefit of interest and parameterized by the time period of interest, since the benefits may vary over time. A basic expression representing the costs and benefits of building a software product line is

$$C_{org}() + C_{cab}() + \sum_{i=1}^{n} (C_{unique}(product_i) + C_{reuse}(product_i)) \quad + \quad \sum_{j=1}^{nbrBenefits} B_{ben_j}(t)$$

where *n* is the number of products in the product line and *nbrBenefits* is the number of separately computed benefit functions. However, SIMPLE's cost and benefit functions can be combined in a variety of ways to model the costs and benefits of specific software product line decision alternatives.

Clements, McGregor, and Cohen provide several scenarios that describe typical product line decision points. They also provide the economic model that can be used to support the decisions in each scenario [Clements 2005a].

SIMPLE does not define specific implementations of the functions. Instead, it only specifies their meaning, leaving the modelers free to provide implementations that take advantage of the data and local knowledge they have about each cost or benefit. The implementations may be based on legacy data or estimation formulas.

Economic modeling can also be used during make/buy/mine/commission analysis. The basic functions can be defined in terms that allow the comparison of long-term cost savings among different development choices. Economic modeling is also useful in conjunction with the "Scoping" and "Market Analysis" practice areas.

**COPLIMO:** The Constructive Product Line Investment Model is based largely on the COCOMO for software cost estimation [Boehm 2004a]. The developers of the COPLIMO studied several product line and reuse efforts in order to establish key parameters that could predict costs. The model includes two components: a cost model for product line development and an annualized post-development, life-cycle extension. While not a business case method, the COPLIMO can be used as a model for gathering financial information that forms the core of a business case. One other noteworthy feature of COPLIMO is its treatment of post-development or sustainment activities that examines such qualities as ease of maintenance, variation mechanisms, and portability.

## Practice Risks

An inadequate business case (or the lack of any business case at all) can set up a product line organization for failure, by inaccurately predicting the outcome of either a product line effort or an individual product launched from within a product line. If the prediction is too optimistic, the organization's investment will not be returned; if it is too pessimistic, the organization will shy away from what might have been a good opportunity. Failure to produce a business case will leave an organization without any way to judge whether the effort was successful. An inadequate business case can result from

- **insufficient data:** It is usually necessary to set cost expectations early and then refine the cost information as the effort progresses. An organization needs time to overcome the "sticker shock" that is usually associated with product line adoption. If sufficient cost information is not developed early, the adoption will likely be on hold while the cost/benefit information is scrutinized and validated. That may cause a loss of momentum or possibly even the disbanding of the project.
- **unreliable historical data:** Most cost development methods rely on good historical data, either from within the organization or from industry. The reliability of the data is essential to justify the approach proposed in the business case. The business case must be able to compare past, current, and projected costs, time to market, market share, and competitor information in order to make the case. While it may be possible to estimate prior results, those estimates will tend to weaken the argument.

- **approaches that fail to work across organizational boundaries:** The business case must be specific to the organization's goals and mission. However, because product line approaches are new and cover a wide range of organizational, technical, and managerial issues, the business case must draw on cross-functional resources from across the organization. That activity will require careful planning and intergroup coordination to meet critical milestones, including budget timetables and personnel availability.
- **uncertain market conditions:** How much will the transition cost? Who will use the core assets? How many products will be needed per year? How long will the product line last? The organization must consider a number of cost factors when developing the business case. While there may be a good basis for estimating the costs of software development, the costs of intangibles such as changing the process from single-system to product line orientation will be difficult to predict and measure. The ability to predict "core asset value" (the overall benefits of core assets to product development) includes making correct assumptions about how product developers will use the core assets. If fewer products than expected make use of the core assets, the overall cost per use will increase and affect the positive ROI and time it takes to reach the break-even point. Management is unlikely to provide continued support without achieving these important goals.
- **management indecision:** The group developing the business case must understand the audience. This audience must include those who can make the final go/no-go decision for proceeding on the proposals contained in the business case. If the business case is presented to the wrong audience, there will be no decision. A business case that does not address the needs of the decision makers results in a no-go decision. Effective communication requires an understanding of the decision makers' value system in terms of the time to market or other financial considerations for commercial organizations. Also, the developers of the business case must determine up front whether the audience wants a range of choices from which to make a selection or a specific decision/policy package on which to base the go/no-go decision. If a decision to adopt the product line approach has been made already, decision makers are more likely to want a set of alternatives from which to choose a specific approach.
- **a shift in organizational goals and needs:** If the goals and needs of the organization have shifted during the preparation of the business case, the results may not be useful or meaningful. If information is presented incorrectly, the business case will have no impact. As a result, there will be either a no-go decision or no decision at all, although facts may justify the business case as presented.

## Further Reading

[Baldwin 2002a]
Baldwin and Clark describe how net options value applies decisions regarding software modularity.

[Boehm 2004a]
Boehm presents the COPLIMO.

[Clements 2002c, p. 226]
In the sidebar "It Takes Two," Clements and Northrop make a close examination of the payoff point for software product lines.

[Cohen 2003a]
Cohen discusses when software product lines pay off, which is at the heart of any business case.

[Ganesan 2006a]
Researchers from the Fraunhofer Institute and Hitachi have shown how to use SIMPLE to calculate reliable ROI predictions in the face of uncertain input values and a core asset base that becomes harder to use as the number of products increases over time.

[Humphrey 2000a]
Watts Humphrey's article provides an example of how to make a business case for process improvement. It assumes that management is thinking strategically and will consider investments that may take a few years to pay off.

[Reifer 1997a]
Reifer provides an excellent set of guidelines for developing a business case to support a reuse effort. Topics include scoping the market, developing the business case, and preparing financial data. This book also includes a sample business case and the steps used to prepare it.

[SEI 2007h]
This Web site provides more information about SIMPLE.

[Weiss 1999a, pp. 45-49]
Weiss and Lai present a short but useful section on building a product line business case in their book *Software Product-Line Engineering*.

## Customer Interface Management

During the development of complex systems, there are dependencies and commitments among the producers of products (including products in intermediate stages of production) and the people for whom these products are intended—that is, the customers.

Customers have requirements for and expectations of what the producers will provide and may communicate them either in a highly structured fashion (such as in a formal requirements document) or in an informal way (such as when an interdisciplinary integrated product team has producers and customers working side by side). In the case of a commercial product produced and mass-marketed for end-user consumption, the "agreement" is the one between the vendor and the perceived demands of the marketplace, possibly gathered through surveys or interviews with proxy groups standing in for the customer community at large.

Understanding and managing the commitments between your organization's producers and customers will require your organization to manage its customer interface. Doing so involves knowing answers to the following questions:

1. Who is involved? Who are the customers or customer representatives? Customers are sometimes represented by individuals who portray particular facets of the customer's overall interests, such as legal, financial, technical, operations, or training. Which groups or individuals are responsible

for interfacing with customers? Typically, they might include marketers, product managers, requirements engineers, architects, or a user group coordinator. Identify exactly who is responsible and clearly define their roles and responsibilities.

2. What information will be communicated and delivered? What are the product offerings, and what variations are available? What are the cost, schedule, and quality benefits of doing business with the organization? What is the strategy for evolving the product(s)?
3. How is the information communicated and delivered? What policies and procedures apply to customer interactions? How are customer requirements to be negotiated and managed? How will a consistent interface with the customer be enforced? How is the customer to be kept informed of schedule, budget, progress, and unexpected problems?

Managing a customer interface also involves ensuring that those in the organization with customer responsibilities are trained properly in their roles and applicable processes.

### Aspects Peculiar to Product Lines

Organizationally, the customer interface must be managed from a multiple-product, multiple-customer perspective as opposed to a single-product, single-customer (or single-customer-base) perspective. This means that those responsible for interacting with a particular customer about a product must act on behalf of both the customer and the product line organization at large.

Over time, customers will want new products or features and ask the product line organization to add to existing products. These requests will have to be evaluated from two dimensions: the feasibility of carrying out the desired changes and the desirability of including them in the current or future development cycles. The product line's scope will address the first aspect, while a business case will address the second. From a strategic perspective, features that have widespread appeal represent tangible opportunities to expand the product line.

These considerations mean that some of the traditional customer-interfacing roles will have to extend their interaction to consider the entire product line. For example, the architect and requirements engineers certainly need to understand the requirements of each individual customer. But they also need to understand them in the context of the overall product line. They may even need to "nudge" a customer's requirements where possible to gain closer alignment with the other products in the family, so as to achieve higher usage of the existing core assets and achieve faster time to delivery and lower cost.

Table 8 identifies a set of product line roles, the activities they typically involve, and some artifacts they commonly use in their customer interface duties.

*Table 8: Typical Product Line Organization Roles Involved in Customer Interface Management*

| **Interfacing Role** | **Typical Activities** | **Example Artifacts** |
|---|---|---|
| marketing, sales, financial, and legal personnel | • Explain product line offerings.<br>• Explore the customers' requirements.<br>• Prepare proposals and conduct preliminary negotiations. | marketing plan, product proposals, product offerings, specifications, other marketing collateral, requirements input to product management plans, upgrade plans, legal and contractual documents |

| | | |
|---|---|---|
| | • Represent the interests of the organization in contract negotiations. | |
| product manager | • Represent the product developer to the customer.<br>• Be responsible for the customer's product and all liaisons with the customer after contract award.<br>• Be responsible for finalizing negotiations and coordinating efforts across the product line organization.<br>• Be responsible for overseeing all the programmatic aspects of the customer interface, including the specification, development, delivery, and acceptance of the customer product. | technical proposals, product specifications, schedule and cost data, work breakdown structure, technical documentation, status/progress reports, action item tracking, issue resolution, product deliverables |
| program manager | • Be responsible for the business success of the product line. | business case, forecast, cost/pricing models |
| senior technical staff members (domain and architecture experts) | • Assist marketers, product managers, and customers in defining the technical specifications of product offerings and negotiating tradeoffs in requirements affecting cost, schedule, and quality. | domain model, functional specifications, architectural views, quality attributes |
| systems/software product engineers | • Deliver and install product upgrades.<br>• Assist customers in resolving operational problems and provide training and technical consultation. | product releases, technical advisories, trouble reports, training courses |
| user-group coordinator | • Work with customer groups to collect product requirements that meet the collective and prioritized needs of users. | customer workshops, strategy sessions, customer operational plans |

For example, those who interact with a customer to sell a product go about their jobs in quite a different way in a product line organization. (In this passage, we'll call these people "marketers," but in some organizations this activity might be done by business development staff, product managers, sales personnel, or others.) In many software organizations, the marketers court customers by promising (and then pricing) the features that the customer requests. In a product line organization, however, the marketers have a responsibility to the entire product line and its customer base. For example, after listening to a customer's requirements, the marketer assigned to that customer must first understand if the desired product is in the product line's defined scope. If so, then all is well, and the marketer can promise delivery with confidence. If not, however, the marketer must interact with the technical staff to assess the proposal's technical feasibility and cost. The response to the customer must now be negotiated based on whether the organization wants to work on a product that is outside of the product line's scope. Perhaps the new product represents a funded opportunity to move into a desirable new business area. Or perhaps the opportunity cost of this new product is too high, and the organization would be better off negotiating contracts more in line with its production capability. In that case, the marketer must now negotiate with the customer about the non-conforming requirements. Or perhaps a customer will be willing to abandon out-of-line requirements once he or she knows their added cost and risk. In any case, the decision should be justified and documented in a business case (see the "Building a Business Case" practice area).

Aspects of the customer/development organization interface necessitated by a product line approach include the following:

- Communicate the product line strategy to the customer. Provide the customer with information and guidance on product features and capabilities, customer-configurable options, and future development goals, objectives, and strategies. Communication involves educating the customer to encourage desired behaviors, helping the customer understand the rationale for deferring product requirements in order to reap cost, schedule, and quality benefits, and moving the customer toward acceptance of a customer relationship that is similar to a commercial, off-the-shelf (COTS) supplier model. The interface with customers becomes market driven and no longer focuses on an individual customer's specialized requirements. Specialized requirements can be accommodated but must be considered individually and have special cost and schedule implications to which both parties must agree.
- Enforce strict product line organizational interfaces and operational procedures (see the "Structuring the Organization" and "Operations" practice areas). In particular, customer requirements for one product cannot be allowed to unilaterally supersede those of other products without considering what is best for the product line overall.
- Encourage customers to choose from standard product offerings, relinquishing some flexibility in return for cost and time-to-market advantages.
- Help customers form user groups to give the *market* a voice and drive requirements for product evolution jointly.
- Institute a protocol for managing customer requirements on a product line (not an individual product) basis, and, through training, compel both the customers and the product line organization's personnel to follow it. The roles, responsibilities, and processes involved with the customer interface should be documented as part of the product line's operational concept (see the "Operations" practice area).

## Application to Core Asset Development

If core assets are produced by a separate organizational entity, in many ways that entity comes to resemble a software development organization of its own, with the need to manage its own customer interface. To a core asset group, the customer is the set of product-building groups, and the same issues apply. Individuals working across the interface must be identified, their roles and responsibilities identified, the interface defined, processes for interaction defined and followed, and training provided.

Some software product line organizations actually make their core asset base "open;" that is, they allow other organizations to use it (for a fee) to build their own products. In this case, the core asset group has a customer interface to the outside world that must be managed along with its interface to its customers inside the organization.

Some of the core assets in the core asset base are aimed at the customer community; for example, marketing literature, sales support, product catalogues, feature descriptions, and cost and schedule information. Other core assets are a direct result of customer input; for example, market analysis, requirements specifications (including the common look and feel of the products), and the product line scope.

## Application to Product Development

The organization's customer interface comes heavily into play in requirements negotiation for a product or set of products. A healthy customer interface will allow a customer to negotiate in the context of the benefits and limitations that a product line approach can provide. From the organization's side, it should be able to offer high-quality products of demonstrated capabilities and performance with predictable delivery dates and predictable costs.

Moreover, a product line organization is in a better position to produce "whole" products that include accompanying artifacts such as robust training materials, support, and documentation (that are backed up by standards and procedures), extensive test cases, and a proven operational track record. Since these associated product artifacts come bundled with the customer's product, from a marketing standpoint, they are touted as being "products with a pedigree" [Moore 1991a].

## Example Practices

**Establish an overarching customer interface process for developing work proposals and negotiating new contracts:** At a minimum, this interface should cover roles and responsibilities including marketing, customer negotiations, contracting, customer liaison during product development, delivery of products and product updates, communications and problem resolution, product support, training activities, and other customer support functions.

A customer's request for changes in a product may or may not be within the product manager's authority to approve. Minimally, the customer's product manager must first determine if the request could be satisfied through existing product line components. If not, the product manager must determine if the product line manager believes the requested capability is likely to be needed in the future by existing or new customers. If that is the case, the capability can be developed with the rigor necessary for reusable components and added to the core asset base for the product line. Typically, this kind of unanticipated product decision would require the reallocation of resources or a downstream adjustment in product enhancement planning. The process needs to include safeguards to ensure that the established policies, procedures, and protocol for customer interactions are followed. This process must address issues such as

- How are the potential architectural ramifications of a customer's requirements taken into account in the requirements negotiation process?
- How are conflicting requirements or priorities reconciled across the products in the product line?

**Provide centralized product support to customers:** This support should include coordinating changes in requirements, providing fixes for operational problems, and managing multiple (new) product releases across the customer base. An essential piece of the customer interface management practice is providing ongoing product support to meet the operational needs of the customer. Such support typically begins with the product line organization that is responsible for helping the customer plan the system installation and ensure that all the required elements are in place. Support is then available for the installation itself. Once the system is operational, ongoing maintenance fixes may be required to stabilize the new system in the customer's operating environment. Ongoing support may also be available to help customers install product enhancements or major upgrades. In the product line organization, some of these responsibilities are centralized so that efforts to fix, test, and release product updates are not replicated. Fixes for one customer are compiled and released for the benefit of all the customers of that system.

Finally, the support organization is in the best position to track customer installations and configurations. This information is useful to the product engineering organization in planning the distribution of new releases, as well as to the marketing organization as a means of identifying opportunities for the sales of upgrades or new systems.

**Establish a user group, or other liaison means, to help the product line organization identify and prioritize its customers' emerging and long-term future needs:** The product line organization should encourage customers to form user groups to jointly manage the evolution of its products and drive product line requirements so that upgrades can be provided to all of them at a lower cost than would be possible individually. In user groups, customers who were once separate entities with no common interest have an opportunity to influence and work out requirements for future product upgrades jointly. From the standpoint of the product line organization, user groups represent an opportunity to maintain a tightly controlled product line capability by providing its customers with a forum for adjudicating their differences and migrating its products as a collection rather than as a set of disjointed products. Both sides benefit: the product line organization can continue to produce products that closely match its product line capabilities, and its customers can acquire their products more economically.

## Practice Risks

Poor customer interface management can undermine the integrity of the product line and jeopardize the organization's market share. If customers are not paid enough attention, the products will fail to meet their needs, and the customers will go elsewhere. If the customer interfaced is not managed at the product line level, the organization will fail to take advantage, in a systematic way, of the customers' inputs. Poor customer interfaces can result from any of the following:

- **The organization fails to recognize the extent of the customer interface and its effects:** A fundamental risk is that an organization will not fully realize the extent of the interface or the extent of the cultural changes that must occur on both sides to establish a healthy and mutually productive customer relationship. This risk could lead to a failure to exploit common requirements, customer alienation, and an eroding customer base.
- **The organization reacts to transitioning to a new business paradigm with discontent and resistance:** Shifting responsibilities and the narrowing focus of new positions may not be welcome changes for many individuals in the product line organization. Satisfying individual customers, possibly at the expense of the product line, is a powerful incentive to undermine the product line approach. The result will be business as usual. Investments in internal promotion and capability development reinforce the organizational commitment to the change and provide team members the opportunity to change with the organization.
- **Marketers promise the world and fail to point out any tradeoffs that are involved:** When pressed by their customers, marketers must learn to point out any key tradeoffs that are involved

so that the customer can make fact-based, objective decisions. Failure to do so will result in dissatisfied customers and (in turn) unhappy product developers.

- **Marketers and product managers are insensitive to specialized customer requirements:** A product line organization must ensure that the pendulum does not swing too far away from individual customers' needs. This is a key dilemma for the product line organization—that is, how to satisfy the specific needs of one customer without significantly impacting the product development momentum within the development group. This problem requires analysis by a cross-functional team to assess the impact of the requirement and make recommendations to management.
- **Customers fail to recognize benefits properly and see only the loss of flexibility:** Some customers will be unwilling to give up the control that they associate with conducting business with a systems organization. They will continue to demand that systems be built to the full desired functionality, regardless of the cost, schedule, or risk benefits that a product line approach delivers. In the face of this inherent conflict, the product line organization and the customer will need to make a decision about the viability of their long-term relationship.
- **The product line organization releases a scheduled product upgrade to all customers that includes unannounced changes that cause sporadic problems for customers:** Forcing unwanted changes on customers will alienate them.
- **The organization fails to enforce the interface:** In organizations with poor product line discipline, customers may have direct access to development staff and may coerce or cajole them into producing specialized features just for them. While such direct connections make those customers very happy and enhance the organization's reputation for responsiveness, they almost always work to the detriment of the larger product line effort. They short-circuit the product planning process and result in work that satisfies small short-term goals at the expense of much larger long-term ones.
- **The customer interface staff is not trained properly:** If representatives are not oriented toward product line operations and miss culture changes, or if representatives are not knowledgeable about the product line strategy and specific product capabilities, the result will be a customer community that is not properly indoctrinated into the benefits of the product line paradigm.
- **Customers with their own agendas dominate user group forums so that other users or customer communities are not heard:** If this is the case, the result will be product line requirements that become skewed artificially in the direction of the dominant customer at the expense of the needs of other customers in the installed base.

## Developing an Acquisition Strategy

Acquisition is the process of obtaining products and services through contracting [Bergey 1999b]. This practice area applies to those organizations that are purchasing or commissioning, rather than developing, at least some of the products, or parts of the products, that they turn out. The growing trend towards outsourcing or "off shoring" makes this practice area much more common than ever before. If your organization has limited resources, you can use acquisition in a variety of ways, from augmenting your own development effort to commissioning the development of an entire product or group of products. Other motivations to use acquisition include being able to

- globally gain access to additional resources on a 24/7 basis to maximize the workforce's availability and shorten the development cycle
- increase competitiveness by leveraging the most economic resources available to perform a specific activity in a timely manner
- take advantage of geographically distributed company resources and their potentially unique perspectives and talents

If you commission an outside source to provide assets for any part of your operation, you will have to incorporate the means of managing the work performed under contract and the resultant contract deliverables. To be effective in contracting, you must develop an acquisition strategy so that you can mitigate the risks associated with acquiring technical products and services from external sources and integrating them into your operations.

Because an acquisition strategy is of great importance to those organizations that primarily acquire rather than develop, this practice area is especially important for government agencies, such as the U.S. Department of Defense (DoD). Although industry may rely heavily on acquisition for obtaining software, corporations are less likely to acquire an entire system. For the DoD, this is closer to the norm. The DoD is an acquisition-based organization.

For acquisition-based organizations, a traditional acquisition approach is to have a separate acquisition for each system and to maintain the system independently throughout its operational life. This is a case of *n* acquisitions for *n* system developments followed by *n* maintenance efforts that may require another multiple-of-*n* acquisitions (because maintenance contracts may have to be rebid several times for long-lived systems) in order to provide ongoing support for the life of the systems.

Developing an acquisition strategy involves analyzing alternative contracting approaches (some of which are offered as example practices), considering pros and cons, and performing tradeoffs. The strategy should address

- establishing the contracting approach and number of contracts needed to satisfy the acquisition requirements
- choosing the contract types, funding alternatives, contract options, and source selection procedures
- building in provisions to provide tasking flexibility
- establishing whether a request for information (RFI) should be used to narrow the field of offerors who could potentially provide the desired products and services
- determining how the continuity of contractual products and services can best be sustained over the projected life cycle
- establishing an approach for obtaining final acquisition approval

For each contract, the strategy should answer the following questions:

- What should be specified in the request for proposal (RFP), which is the initial solicitation by the acquiring organization to potential contractors?

- What should be included in the statement of work (SOW), which defines the work and work products that will be provided by the successful bidder?
- What technical evaluation criteria should be used to choose among competing bidders and judge the quality of the delivered products?
- What kinds of incentives are appropriate?
- What deliverables should be required?
- What data rights should be incorporated, and, if open source software is involved, how will the data rights to the system be protected?
- What means shall be included to adequately monitor the contractor's technical performance and progress and promote accountability?

Multiple contractors may be involved in acquisitions that incorporate competitive runoffs, multiple suppliers, or teaming arrangements.

An acquisition plan is the artifact used to document the acquisition strategy. The plan should also record the costs, risks, and considered alternatives to the adopted strategy.

## Aspects Peculiar to Product Lines

A product line acquisition strategy is a plan of action for achieving a specific product line goal or result through contracting for products and services. In the case of a software product line, the types of software *products* acquired through contracting include core assets and derivative products built from those core assets. Potential software *services* include elements of core asset development, product development, and management. Acquiring services means contractually engaging an identifiable task rather than furnishing an end product.

If your make/buy/mine/commission analysis returned "commission" for any product line asset, you have made a decision to acquire that asset and therefore need an acquisition strategy. Even if you rarely acquire software, you still need an acquisition strategy for those occasions when acquisition is your chosen (or only) option.

Acquisition for product lines is typically structured much differently than acquisition for single systems. First, the acquisition of core assets is usually the result of a contracting effort apart from the acquisition of products. Second, the fundamental role of architecture in a product line imparts opportunities for contracting flexibility. An umbrella acquisition might provide the entire product line production capability—architecture, infrastructure, and core assets—or the production capability might be acquired in pieces. The architecture itself might be acquired as an independent step. The acquisition of software components or services (whose interfaces and interoperating requirements will have been specified by the architecture) might be distributed among several suppliers. Products built from the core assets might be acquired individually or under contract for an entire set. A small number of follow-on acquisitions can accommodate ongoing product development and support for sustaining and enhancing the core assets and products over the life of the product line.

Plan your acquisitions early. The contracting process is often arduous and can consume an enormous amount of time before a contract is in place and operative. Plus, an acquisition strategy cannot be developed in isolation. It exists to serve the product line goals and interacts with the results of other practice areas (named below), some of which inform us about what we wish to acquire. Accordingly, you will need to allow sufficient time to coordinate and interact with those carrying out the work of these other practice areas.

If acquisition will fulfill a major role in your product line, you should form a small acquisition team to develop and implement an acquisition strategy. Begin by drafting a charter to empower the team, define roles and responsibilities clearly, and ensure the participation of key stakeholders. The team leader should come from the product line organization. Team members should include a contract negotiator, a representative from the product line organization, representatives from participating product groups, individuals with significant acquisition experience, and key product line stakeholders who have a vested interest in the acquisition. Make sure that you

- select motivated individuals who have a “can-do” attitude
- keep the team together through contract award and the start-up of operations to ensure the accountability and continuity of effort
- obtain the early buy-in of those responsible for approving the acquisition plan before committing to a particular strategy

The acquisition strategy team should begin by understanding the organization’s concept of how the product line will operate and the role that acquisition will fill. Therefore, the team should coordinate with those carrying out the activities in practice areas in which these issues will be decided. These practice areas include

- “Scoping,” in which the product line is defined
- “Funding,” in which the strategy for paying for the core assets is determined
- “Architecture Definition,” in which a large segment of the necessary core assets is first identified
- “Make/Buy/Mine/Commission Analysis,” in which decisions are made about which artifacts are acquired externally
- “Technology Forecasting,” which may provide insights about what assets might be acquirable in the future

A product line approach is a natural fit for specifying and coordinating efforts across a distributed or geographically dispersed workforce. Facets of a product line approach that support workforce distribution include

- an overarching concept of operations (CONOPS) that unifies product line activities according to predefined roles and responsibilities and specified practices
- an architecture-driven approach that provides a basis for organizing distributed development
- a generic production plan that prescribes how individual core assets are combined to form derivative products

## Application to Core Asset Development

Acquisition can serve several roles in obtaining all, or part, of the core asset base. They include the commissioning of one or more contractors to

- develop the product line architecture
- develop other core assets (for example, components and supporting artifacts)
- mine legacy assets for inclusion in the asset base
- manage, sustain, upgrade, and enhance the existing core asset base and provide support to product developers

Core asset acquisition may also involve the acquisition of contractor services, such as scoping, domain analysis, configuration management (CM), testing, and training. An acquisition organization can fill these core asset development needs and/or acquire these services through a single acquisition or multiple acquisitions involving one or more contracts (or contract options) that run sequentially, run concurrently, or overlap.

If an organization is going to commission the development of core asset software components, an architecture will have to be specified, because the architecture places constraints on the components or services that will be included in the core asset base. The architecture also determines which components are common across all products (or at least across subsets of the product line) and defines the necessary variations among instances of those components. Since the architecture plays such a pivotal role in product line operations and constitutes a strategic competency area, the business impact and necessity of commissioning the development of the product line architecture by an external party should be carefully considered.

## Application to Product Development

Acquisition is also an effective means of obtaining new products (or parts of products) in the product line. Acquisition can play several roles, including the commissioning of one or more contractors to

- develop a specific product or a set of new products using the core asset base and following the production plan
- maintain, upgrade, and enhance a product or set of products
- incorporate or evaluate new releases of core assets in a product or set of products to promote product compatibility with the current core asset base and the overall integrity of the product line
- provide new assets (created in the course of product development or product enhancement) to be evaluated as candidates for potential inclusion in the core asset base

Product acquisition may also involve the acquisition of contractor services, such as requirements engineering, CM, architecture evaluation, software system integration, and testing.

A commercial organization must carefully consider how to protect intellectual property rights when commissioning the development of products from its core asset base. Lack of care could result in the supplier becoming a competitor. One way to reduce this risk would be to commission the development of product-unique parts or to contract for specific product-creation services. One of the example practices below provides guidance in this area.

## Example Practices

**Considering core competencies and strategic directions:** For commercial organizations in particular, selecting an acquisition strategy should be dependent on what core competencies the organization possesses and the strategic business impact of acquiring products and services from external sources. The criteria embodied in Figure 13 provide insight into the considerations that should potentially govern whether a product line operation should be included in the acquisition.

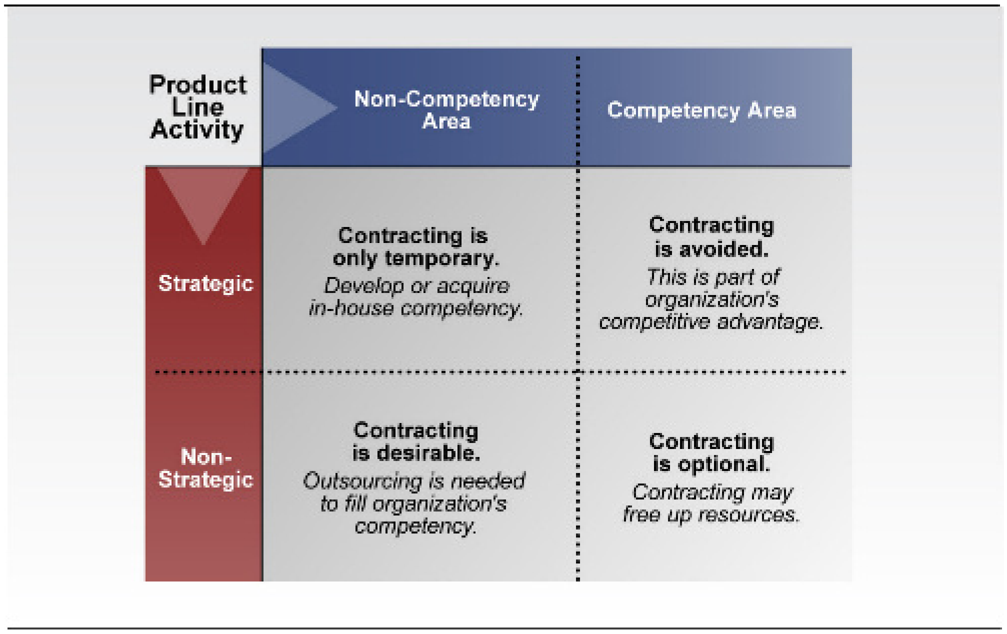

Product Line Activity
Non-Competency Area
Competency Area
Strategic
Non-Strategic
Contracting is only temporary.
Develop or acquire in-house competency.
Contracting is avoided.
This is part of organization's competitive advantage.
Contracting is desirable.
Outsourcing is needed to fill organization's competency.
Contracting is optional.
Contracting may free up resources.

*Figure 13: Criteria for Outsourcing a Particular Product Line Activity*

The considerations for these criteria are

- Certain product line activities may have a strategic impact on an organization's business.
- Every organization has areas of competency and non-competency.
- An organization must be competent in areas that are important to its strategic business interests.
- An organization should avoid contracting out any activity that could negatively impact an organization's strategic business interests.

Clemons and colleagues provide in-depth insight into the issues that inform and motivate such strategic sourcing decisions and their associated risks [Clemons 1997a]. The criteria shown in Figure 13 are useful for establishing what product line activities are candidates for outsourcing. The selected activities become the basis for developing a comprehensive acquisition strategy that can achieve the desired end state.

**Setting the acquisition context with the SEI *Adoption Factory* pattern:** Even if an organization relies heavily on acquisition to achieve its goals, it will still have a major role in the execution of many of the product line practices. In setting the context for the product line, the acquisition organization will have to specify what constitutes the product line and establish structures and practices that parallel those of an organization that does most of its work in-house—a key difference being the emphasis on "doing" versus "overseeing the doing."

The "Launching and Institutionalizing" practice area contains a discussion of the Adoption Factory pattern, which provides a general roadmap for product line adoption [Northrop 2004a]. This pattern is a useful guide for helping acquisition organizations determine what they should do first. The following are some of the important early tasks listed in the pattern's Establish Context phase:

- determining what should be in the product line. (See the "Scoping," "Building a Business Case," "Market Analysis," "Understanding Relevant Domains," and "Technology Forecasting" practice areas.)
- establishing product line acquisition processes. (See the "Process Discipline" practice area.)
- preparing the acquisition organization for product line operations. (See the "Launching and Institutionalizing," "Funding," "Structuring the Organization," "Operations," "Organizational Planning," "Customer Interface Management," "Organizational Risk Management," and "Training" practice areas.)

**Putting architecture first**: One acquisition strategy, which is most applicable to government agencies that rely very heavily on acquisition, involves procuring only an architecture in the first stage, procuring other core assets in the second stage, and procuring products (built from the core asset base) in the third stage. The first stage would then focus on devising a strategy for acquiring the architecture. Listed below are four potential strategies for architecture acquisition that are distinguished by the source of the architecture.[9]

1. **systems architect:** A contractor is hired to develop the architecture for the system, but the contractor does not build the system or components. The acquiring organization funds, owns, and controls the product line architecture. Another contractor is hired for the implementation of the architecture. This strategy helps to curtail the risk of getting an architecture that does not fulfill program needs, because the program retains control over the architectural development. However, there is still a risk in terms of whether the architecture will be implemented correctly when the system is built, as well as other risks in terms of cost and schedule adherence.
2. **standards group:** An architecture that is either built by a "standards group" or conforms to established standards is acquired. Industry and/or government collaboration creates a public architecture. The acquiring organization influences but does not control or own the product line architecture.
3. **single contractor:** A single contractor is commissioned to develop the architecture. The contract is put up for bids, and contractor selection is based only on the architecture portion of the system

[9] Fisher, M. CECOM *Architecture for Practitioners Course*. SEI course given in 1998.

design. However, the same contractor may also end up developing components and/or the system under the contract. The contractor supplies the architecture and ownership, and control is negotiated.

4. **collaborating contractors:** A contract is developed that requires a group of contractors to collaborate on developing an architecture that they all can use later. In addition, each of the contractors is given a contract to develop and maintain some of the system's components. Ownership of the architecture is usually shared among the development contractors, with the acquiring organization holding the licensing rights. The acquiring organization funds joint development and manages the architecture requirements.

**Other acquisition strategies:** There are several other acquisition strategies for moving toward a product line capability. They are distinguished by the initial product line capability that is desired. They include the acquisition of

- a software architecture for the product line (discussed above)
- a system architecture (similar to that discussed above)
- an architecture and set of components (and related artifacts) that conform to it
- a product and some set of core assets

The last strategy listed above, which focuses on product development, also results in the acquisition of an architecture, a set of components and their supporting artifacts, and a product built using these core assets. This strategy is an extension of the third strategy listed above, which reduces the risk of architectural and component incompatibilities. The quality of the architecture and components is demonstrated more thoroughly by building an actual system based on the core assets. Also, this strategy aligns with the natural iterative learning that takes place in establishing a product line. Proving that the core assets can be used to build a system provides valuable credibility for them.

A significant variation on this "core assets plus product" strategy is to acquire an additional system—a second product—that will reuse the core assets. This approach allows the program to reap the benefits from the investment in building the core assets.

## Practice Risks

Introducing acquisition into the equation clearly provides an organization having limited resources with an effective means of pursuing a product line approach and tapping skills and resources that would otherwise be unavailable. However, acquisition also has its attendant risks by virtue of introducing a new, and sometimes arduous, paradigm for managing the products and services that are acquired through contracting. A poor acquisition strategy will result in contracts being let that are not in the best interests of the acquiring organization. In the worst case, the goods delivered do not meet the needs for which they were acquired, and the resources (not the least of which are the time and effort spent on the contract and the time spent waiting for the goods) are wasted. Deadlines will be missed as the organization scrambles to recover. A poor acquisition strategy can result from any of the following factors:

- **Poor communication and contract oversight:** The more contractors involved, the greater the risk and the need for robust communication and effective contract management.

- **Failure to accommodate iteration:** Iteration is a handmaiden to product line practice. It occurs throughout the operational picture, especially during a product line's start-up and evolution phases. Contracting, however, is not designed to be iterative. Thus, creative means that indicate a prescribed protocol for managing contracts and accepting contract deliverables must be preplanned and included in contracts. Such means include specifying interim or partial deliverables, conducting technical interchange meetings (TIMs), including in-progress development checkpoints for events such as architecture evaluations, specifying *services* to enable contractor participation in CM control board meetings, software integration efforts, and technical consultations to assist product development groups.
- **Limited management visibility:** Contracting results in a division of management responsibilities that often reduces visibility into the progress and status of the work being performed under contract. Formulating a suitable set of product line measures to monitor the progress of core asset and product development efforts is one means of obtaining the needed visibility. Visibility and insight into the progress of the work and the underlying technical problems are more critical in a product line approach, because they involve more than just an "end-product view" of systems development.
- **Failure to account for evolving requirements:** The nature of a product line is to manage the commonality and variability of products by means of a requirements-engineering-change management process. Contracting, by nature, works best when the requirements are fixed. Again, creative techniques can be employed to accommodate the management of "evolving requirements" and to mitigate the impact on the contractual tasks. Such management and mitigation are especially important in the core asset sustainment and process refinement phases of product line operations.
- **Failure to account for liability:** In a typical single-system acquisition, the contractor is totally responsible for ensuring that the system complies with the contractual requirements. A product line approach often involves multiple contractors, which raises the question of how liability issues involving the efficacy of an architecture and other core assets will be handled. Liability issues can be especially acute if flaws (including documentation errors) are not discovered until much later in product development.
- **Failure to pin down the roles and responsibilities of the acquirer and contractor:** For what things should the acquirer be responsible? For what things should the contractor be responsible? Does the summation of these responsibilities constitute a cohesive approach that covers all the aspects of product line operations? Letting responsibilities fall through the cracks will result in lost time and increased expense that will need to be recovered when the oversight is discovered.
- **Failure to consider ownership and data rights:** If the prescribed data rights are not consistent with the envisioned product line operations and adoptable by the acquirer and contractor, the product line may well be prevented from growing and evolving.
- **Failure to enforce architecture compliance:** What form will the architecture take? Is there a contractual means of verifying that product development is compliant with the prescribed architecture? Failure to ensure architectural compliance in delivered components may result in a set of core assets that will not support a product line.

- **Failure to consider core asset sustainment:** Is there a contractual means of sustaining and evolving the architecture and other core assets? Failure to account for evolution will cause the premature death of the product line.
- **Failure to consider product development support:** Is there a contractual means of providing core asset customization and technical assistance to product developers? If not, the core assets may not be usable in practice.
- **Failure to consider product development:** Is a contractual means in place for projects to commission a product line contractor to develop a product, *or* is there a way for a project's own contractor to obtain core assets and technical assistance for product development?
- **Failure to consider the coupling of contract deliverables:** Is there a contractual means of accommodating product upgrades as changes are incorporated in the core asset base? If not, the product line may produce products that are, in fact, one-of-a-kind systems that drift out of alignment with the core assets.
- **Failure to ensure continuity of support:** Is there a means of ensuring the continuity of acquisition support over the life of the product line?

Tompkins describes 40 major risks when you take the following steps to outsource: strategy, selection, implementation, and management [Tompkins 2004a]. These risks are especially applicable to commercial organizations interested in avoiding the pitfalls and realizing the benefits of outsourcing.

### Further Reading

[Bergey 1999b]
Bergey, Fisher, and Jones give a nice overview of the U.S. Department of Defense acquisition environment interpreted for software product lines.

[Bergey 2004a]
Bergey and colleagues provide a companion document to this framework for government organizations that routinely acquire rather than develop their software. This document provides augmented information for acquisition organizations on the essentials of software product lines. In addition, it serves as an interpretation of the framework from a strictly government acquisition perspective.

## Funding

The activities involved in any software development effort have to be financed; this practice area addresses how. Funding sources and models vary according to organizational culture and the nature of the software product being developed. If multiple copies of the software product are to be marketed, the organization usually appropriates development funds to a business unit, or the business unit appropriates its own funds. New products often get funded initially out of research and development allocations. If a product is being made specifically to serve the needs of one customer, the customer usually provides the funds. The funding of product maintenance is often dealt with separately and may come from a source other than that of the development financing. Whatever the source, somehow the funds are procured to support what it takes to develop and then evolve the software product. Good estimates are required so that an adequate amount is allocated, thereby providing a stable funding source through product completion.

## Aspects Peculiar to Product Lines

A software product line requires funding to get it off the ground. Investment is required for any technology change. In the case of product lines, funding is needed to prepare the organization for a product line approach and may involve training, different development processes, different management practices, and so forth. Funding is also needed to establish a core asset base, to perform analyses (such as achieving an understanding of the relevant domains, scoping, requirements engineering, architecture definition, required variation mechanisms, and so on), and to establish a production infrastructure in accordance with the organization's product line adoption plan and concept of operations (CONOPS). This funding must be sufficient for the core assets to be of high quality and have the appropriate applicability. Once the product line is up and running, it must be sustained and evolved. The core assets must be kept current; new assets may need to be developed; the analyses must be updated; the infrastructure must be modernized. Funding must be stable and enduring so that the core assets can be maintained and the associated product line practices and tools can be supported and improved.

Depending on the approach taken—proactive, reactive, or incremental—different funding profiles are required over time. Once a product line is fully operational, development costs can be apportioned to each new product in a relatively straightforward manner. Often, the key funding question, though, for a product line organization is how to fund the core assets that will be used across several products—most of which will probably be created long after the core assets are initially put in place.

The required funding's magnitude and profile over time should be defined in a business case for the product line. Since a business case explains how the organization will make money by adopting the product line strategy, it must also explain what the organization must invest (and when) to realize the projected payoff. (See the "Building a Business Case" practice area.) The goal of the "Funding" practice area is continually evaluating and planning how the funds needed to develop and sustain the product line will be obtained commensurate with how the scope, business case, and CONOPS evolve. Funding involves evaluating and deciding what strategies can be employed to equitably share/distribute the cost of developing a product line capability and sustaining (and evolving) its ongoing operation.

Transitioning to a product line operation most often occurs in parallel with ongoing operations, because few organizations have the luxury of stopping either in-progress or planned developments while they change course to adopt a product line approach. As a result, new or innovative sources of funding for the organization are often required to support the launch and evolution of the product line.

Although funding may seem to be a pedestrian or esoteric practice area, it has far-reaching consequences for an organization that is relying on a software product line approach. Cummins, Inc., for example, used a funding model to insure the adoption of core assets by business units that otherwise may not have been inclined to participate in the product line effort [Clements 2002c, Ch. 9].

## Application to Core Asset Development

No matter what kind of product line approach an organization takes, it needs to provide funds for managing, developing, and sustaining the core assets as they evolve over the life of the product line. These assets include a broad spectrum of artifacts (requirements, business case, scope, plans, architecture, components, tests) and other elements of the production infrastructure (equipment, processes and tools) that support their development, usage, maintenance, and evolution. The funding required for core asset development can be broken down into planning and analysis, asset development, infrastructure development, and product line sustainment and evolution, corresponding to the first four activities included in Figure 14. The available funding and the funding model chosen influence the production strategy, which consequently influences the production method and ultimately the production plan.

## Application to Product Development

Once a product line is operational and products are being produced on a steady basis, product development costs must be covered. They are usually apportioned to each new product in accordance with the requirements negotiation and technical planning process. The initial funding needs to cover all aspects of managing and developing the particular product in accordance with the production plan and product line CONOPS.

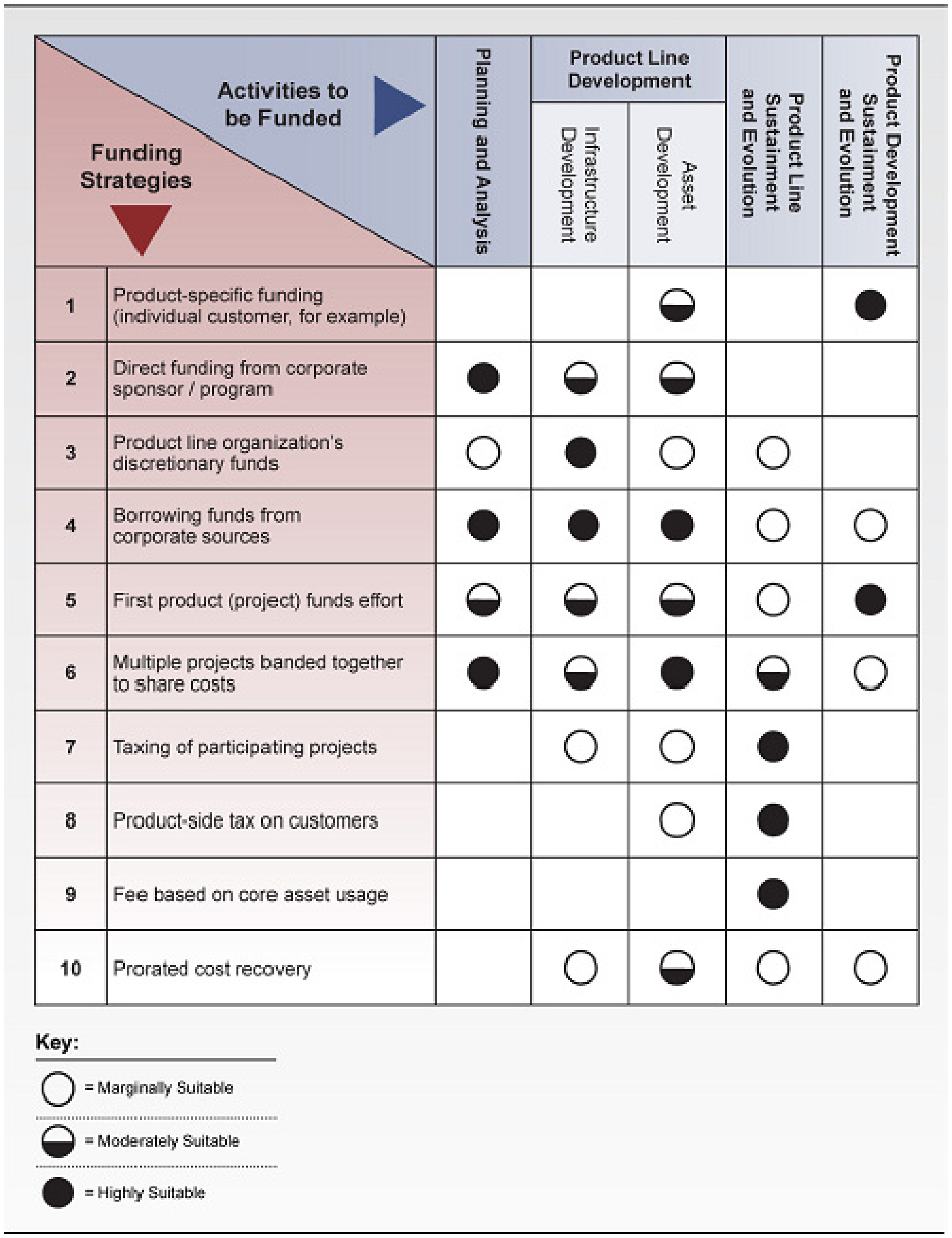
Typically, product development activities are funded predominantly out of product-specific funds. Consequently, the development costs are often not included in estimates of the funding required for developing a product line capability. That does not mean that these estimates are unnecessary or unimportant. On the contrary, funding estimates are a very important (and essential) element in developing a business case for adopting a product line approach and determining the projected return on investment (ROI). Occasionally, the cost of developing the “first product” is considered (and paid for) as part of the cost of developing a product line. In any event, participating projects need a cost (and schedule) estimate so that they can submit a budget for the funds they will require for their product development based on the product line approach.

Similarly, funds for sustaining and evolving the products once they are fielded are also typically obtained from project-specific sources. The benefit that these product development groups realize is that the sustainment costs are substantially lower because the individual products are part of a family of products that share a large number of core assets that are sustained (and enhanced) by the product line organization. Moreover, because the reliability of these core assets is improved as a result of this centralized sustainment effort, the cost of sustaining an individual product is lower.

The funding that is required for product development can be broken down into product development, sustainment, and evolution, corresponding to the last set of activities listed in Figure 14.

## Example Practices

Funding strategies and sources depend on the fiscal infrastructure of the product line enterprise. This infrastructure includes the organizational structure, the mission and functions of the organizational units, the amount and type of funds allocated to these units, and the policies and procedures they must follow to plan and obtain funding. There may be instances in which an existing fiscal infrastructure may not be sufficiently flexible to accommodate unique funding requirements for product lines. Figure 14 identifies potential funding strategies and sources that may be employed to share/distribute equitably the cost of developing a product line capability and sustaining its ongoing operation. These strategies and sources are a representative set that, when adapted and taken in combination, can affect a complete funding infrastructure for a product line initiative.



| Funding Strategies / Activities to be Funded | | Planning and Analysis | Product Line Development | | Product Line Sustainment and Evolution | Product Development Sustainment and Evolution |
|---|---|---|---|---|---|---|
| | | | Infrastructure Development | Asset Development | | |
| 1 | Product-specific funding (individual customer, for example) | | | ◒ | | ● |
| 2 | Direct funding from corporate sponsor / program | ● | ◒ | ◒ | | |
| 3 | Product line organization's discretionary funds | ○ | ● | ○ | ○ | |
| 4 | Borrowing funds from corporate sources | ● | ● | ● | ○ | ○ |
| 5 | First product (project) funds effort | ◒ | ◒ | ◒ | ○ | ● |
| 6 | Multiple projects banded together to share costs | ● | ◒ | ● | ◒ | ○ |
| 7 | Taxing of participating projects | | ○ | ○ | ● | |
| 8 | Product-side tax on customers | | | ○ | ● | |
| 9 | Fee based on core asset usage | | | | ● | |
| 10 | Prorated cost recovery | | ○ | ◒ | ○ | ○ |

Key:

○ = Marginally Suitable

◒ = Moderately Suitable

● = Highly Suitable

*Figure 14: General Applicability of Funding Strategies to Product Line Activities*

The Xs shown in Figure 14 indicate how applicable an identified strategy is to the funding of the indicated activity. Three Xs indicate that the strategy is considered highly suitable, two Xs indicate that it is moderately suitable, and one X indicates that it may be only marginally suitable.

The applicability (suitability) of a particular strategy in a given organizational setting depends on that organization's culture, fiscal infrastructure, and strategic goals and objectives. Use this table as a starting point for your organization.

Each funding strategy listed in Figure 14 is summarized below.

**Product-specific funding (for an individual customer, for example):** In this strategy, each product project provides the funding that is needed for the identified product line activities. While product-specific funding is the predominant means of funding product development, it may be used to fund core asset development as well. Although product projects are often viewed (and properly so) as a primary source of funding, they may be reluctant to pay for aspects of product line operations that they feel are the responsibility of the parent product line organization. As indicated in the table, this funding reluctance may extend to elements of general product line planning and analysis, infrastructure support, and sustainment, because they will also be of direct benefit to other projects involved in the product line. In these areas, other funding strategies are often more effective and should be considered carefully.

**Direct funding from corporate sponsor/program:** This strategy is based on having the corporate-level/program sponsor selectively fund elements of the product line, especially those in the launch of the product line, that are related to the planning and development of an initial set of core assets and the necessary production infrastructure.

**Product line organization's discretionary funds:** This strategy involves using discretionary funds, such as research and development monies, that are directly under the control of the product line organization to offset the greater up-front costs of developing a product line. If the organization responsible for managing and implementing the product line does not have its own discretionary funds (apart from the sponsoring or parent organization), this strategy, in effect, is no different than the "direct funding from corporate sponsor/program" strategy.

**Borrowing funds from corporate sources:** This strategy is analogous to taking out a bridge loan or a mortgage. It involves borrowing product line funds up front and negotiating a suitable payback plan. The funding could cover the launching of the product line through core asset development and development of the first product or another negotiated milestone. The terms might include deferring any payback until the delivery of the first product with a fixed payment schedule or incremental payments tied to an index that is commensurate with product line maturation and fiscal stability. As in a conventional loan, the payoff amount would include cumulative interest over the life of the loan.

**First product project funds effort:** In this strategy, the first product project, in addition to funding its own product development, agrees to provide the funding for other designated activities such as product line planning and analysis, infrastructure establishment, core asset development, sustainment, and evolution. Under this strategy, the extent of the funding provided by the first project may be limited to establishing an initial product line capability, after which other sources of funding are used to sustain and evolve the product line.

**Multiple projects banded together to share costs:** In this strategy, multiple product development groups (projects) agree to form an alliance and jointly fund the cost of developing a product line capability that may potentially extend to funding all product line operations including product line sustainment and evolution. Even in this approach, however, each project is usually responsible for funding its own product development part of the effort. This approach is similar to the one that CelsiusTech took in its product line development initiative: it pooled contract monies from individual customer projects and collectively developed a family of products [Brownsword 1996a]. Another example is the Owen cooperative model from Hewlett-Packard [Toft 2000a].

**Taxing of participating projects:** This strategy involves funding selected elements of the product line by levying a tax on each participating product development group (project). This taxing strategy can use a flat tax or a prorated tax that is based on some particular product attribute (such as product funds, project size, or estimated number of lines of code). The "product-side tax on customers" and "fee based on core asset usage" strategies described below can be viewed as special cases of a taxing strategy.

**Product-side tax on customers:** In this strategy, a surcharge is assessed to fund selectively designated product line activities other than product development. This surcharge is then budgeted into the total estimated cost of developing a product for the customer based on a product line approach.

**Fee based on core asset usage:** This strategy involves charging projects a fee proportional to their usage of the core assets in their product development and/or end products. This strategy is similar to enacting a license fee for using a commercial, off-the-shelf (COTS) product. In fact, in cases in which there are multiple instances of the same product (for example, an application that is operational in many different aircraft), the product line organization may charge a set fee for every copy made or a licensing fee corresponding to the projects' usage of the core assets. Charging such fees is one possible means of obtaining funds for sustaining product line operations.

**Prorated cost recovery:** The object of this strategy is to have the projects that have benefited from the product line pay back their fair share of the costs of any software development efforts or services that the product line organization performed on their behalf. This strategy could be extended to include prorating all of, or just elements of, the total cost of sustaining product line operations among the participating project/product developers.

## Practice Risks

Inadequate attention to the funding model for a product line will result in a core asset base and products whose owners compete in unhealthy ways for the finite resources available. The result will result be poor quality on both sides and probably resentment across the divide. A poor funding model can result from any of the following factors:

- **inflexibility of the organization's fiscal infrastructure:** Each organization's fiscal infrastructure may not be immediately adaptable to the funding of a product line approach. As such, cultural and infrastructure changes may have to be planned and implemented. Such an implementation may take an inordinate amount of time and effort to modify, especially if the infrastructure involves regulations and statutes. Because of such financial barriers, the product line approach may not be initiated, or there may be ongoing contention for funds to sustain and evolve the product line.

- **waning management commitment:** Management's commitment to stay the course and provide adequate funding and resources until the product line reaches critical mass and can sustain itself is key to success. Accordingly, managers must treat the funding of a product line as a longer term, strategic investment that is essential to providing the organization with the means and agility to deliver new products faster and cheaper. Management is responsible for ensuring that short-term crises, competing project demands, and other changes that have the potential to impact funding do not perturb the product line effort. This risk extends to convincing key technical people and their direct managers that funding software product line efforts should, within reason, take precedence over current project demands.
- **externally imposed fiscal constraints:** Organizational cuts and cost-saving mandates can limit the ability to fund new approaches, such as core asset development. Government agencies are especially prone to the annual "battle of the budget" and changes in fiscal policies, but commercial organizations are not immune. These yearly upheavals bring with them the risk of limiting sustaining funds for a product line.
- **lack of strategic focus:** Product line initiatives require strategic planning. Organizations have to overcome a "research and development mind-set" in which funds are spent on a small exploratory effort over two or three years without having plans for how this will evolve into a way of doing business that involves everybody. Moreover, key technical people, or their direct managers, need to be convinced of the priority of funded product line efforts. One of the sources of major frustration and deep-seated resentment is a funding model that lets recalcitrant parts of the organization continue on their one-product-at-a-time way, effectively undercutting the efforts of core asset group that is struggling and lacks the necessary resources.
- **inadequate funding:** If the funding allocated for a product line approach is insufficient, the funds could be spent with little residual benefits. That is, if an organization values only product development and is only paying lip service to product lines, any effort toward establishing a product line will be wasted. Unfortunately, such an occurrence may forever polarize the organization and the workforce against any future consideration of adopting a product line approach.

## Launching and Institutionalizing

This practice area is about organizational change.

Change projects are undertaken to help organizations prepare themselves to adopt a new technology or a new way of doing business. These projects are highly dependent on the context of the organization; an invariant sequence of steps to execute the project is inappropriate. Successful change projects take into account not only the specific technology involved but also the human aspects of change.

Organizational change involves an assessment of the current state, an articulation of the desired state, and an assessment of the gulf between the two. After that, solution strategies for bridging the gulf can be crafted, tried out, and then scaled up. Lessons learned along the way can help refine your understanding of the current state, the desired state, or the intended solutions.

### Aspects Peculiar to Product Lines

The change being launched and institutionalized is of course the software product line approach. This practice area is relevant whether an organization is starting a product line effort for the first time or trying to expand and/or improve the ongoing product line effort. Launching and institutionalizing a product line is somewhat different in that it is a practice area about applying the other practice areas, as appropriate to the needs and capabilities of an organization.

Launching and institutionalizing a product line differs in the particulars from other change projects because product line adoption involves technology and business change. Launching a product line is about the judicious and timely adoption of product line practices. By factoring in a characterization of your individual situation, the part of the product line effort you want to accomplish, the groupings of practice areas that meet your individual needs, and a dynamic view of how the practice areas in that grouping interact to help you accomplish your goals, you can bring the practice areas to bear on your situation most effectively.

The end game of product line adoption is to have an operational product line. A product line adoption plan must be created to describe how product line practices will be appropriately rolled out across the organization. Depending on the starting point of the organization, this plan[10] may provide for the definition of processes, the initiation of practices, the selection and implementation of pilots, or the engineering of a product line. If the intention is that the entire organization will eventually adopt a product line approach, the plan should address the entire organization. For example, suppose that a chosen pilot project involves only one part of the organization that eventually will make the transition. After all, that is one of the benefits of a pilot project: it does not involve the entire organization, so missteps and early mistakes are confined to a small effort. What, then, do the other parts of the organization do while the pilot is underway? If they do nothing, it’s business as usual for them, and in more than one organization we’ve seen, that has spelled trouble. Those people may feel left out and disenfranchised, and their support for the transition to product lines may suffer as a result. But an adoption plan that applies to the organization (a project, business unit, or corporation) as a whole may be the solution. Even if a group is not participating actively in the pilot project, its members can serve as reviewers, receive training, practice building a business case or scope definition, or be assigned process improvement activities to shore up their capabilities for when they will join the product line. A product line adoption plan—the result of the launching effort—is the master plan showing how all parts of the organization adopt product line capabilities, perhaps in a highly staged manner.

Institutionalizing product lines requires that the organization consistently use product line practices to achieve its business goals. Product lines become community practice. To do that, an organization must

- have

[10] For a complex adoption, it is often better to address details in subordinate action plans.

    - a core asset base
    - supportive processes and organizational structures
- develop products from that asset base in a way that achieves business goals
- prepare itself to institutionalize product line practices

### Application to Core Asset Development

Launching a product line will, of course, kick off the core asset development effort, but more is required to ensure the proper organizational context for core asset development. Such activities as funding, training, and structuring the organization will likely be part of launching. In fact, a product line adoption plan may be one of the first core assets that an embryonic product line effort will develop.

Institutionalizing a product line involves improving the processes that are associated with building, maintaining, and evolving the core assets and making those processes part of standard organizational practice.

### Application to Product Development

Launching a product line often involves choosing a pilot project or two with which to demonstrate the product line activities. (For information about choosing pilot projects, see the “Example Practices” section below.) These pilot projects should, if possible, yield marketable products, which will enhance the fidelity of the demonstration and subject the core assets used in their construction to real-life constraints. Institutionalizing a product line means making the development of products routine and predictable while still meeting the organization’s product line goals. The achievement of this steady state is a signal that product line practice has, in fact, been institutionalized.

### Example Practices

Launching and institutionalizing a software product line is a matter of orchestrating the activities of all the applicable practice areas over time. Your organization’s specific launching strategy will be unique. However, many organizations have had success following the SEI *Adoption Factory* pattern in conjunction with a technology change model, such as the SEI Initiating, Diagnosing, Establishing, Acting, Learning (IDEAL) model. (This pattern and model are both described below.) Beyond that, the specific practices discussed below will suggest some additional approaches for bringing your organization up to product line speed.

**Use the Adoption Factory pattern:** Product line practice patterns deal with applying practice areas in a way that is most relevant to the organization’s situation [Clements 2002c, Ch. 7]. Finding (or inventing) the appropriate patterns is, in many ways, the essence of launching and institutionalizing a software product line. The Adoption Factory pattern can serve as a generic roadmap for product line adoption [Northrop 2004a].

The *phases and focus areas* view of the Adoption Factory pattern in Figure 15 illustrates the dynamic structure of the pattern as three columns, corresponding to the temporal phases of product line adoption, and three rows, corresponding to the focus areas for certain patterns and practice areas. The three temporal phases are

1. **Establish Context:** which involves paving the way for product line adoption
2. **Establish Production Capability:** which involves developing the core asset base and the production infrastructure and managing those efforts at the project and cross-project levels
3. **Operate Product Line:** which involves using the core asset base to efficiently build products and monitor and improve the product line operation

The three focus areas are

1. **Product:** which involves those activities for defining and developing products and their common assets
2. **Process:** which involves the underlying processes and production infrastructure necessary to adopt a product line approach
3. **Organization:** which involves the management practices and activities necessary to adopt a product line approach and operate a software product line

Within each of the nine cells created by the intersection of a phase and focus area are the intuitively named subpatterns[11] [Clements 2002c, Ch. 7] that make up the Adoption Factory pattern.

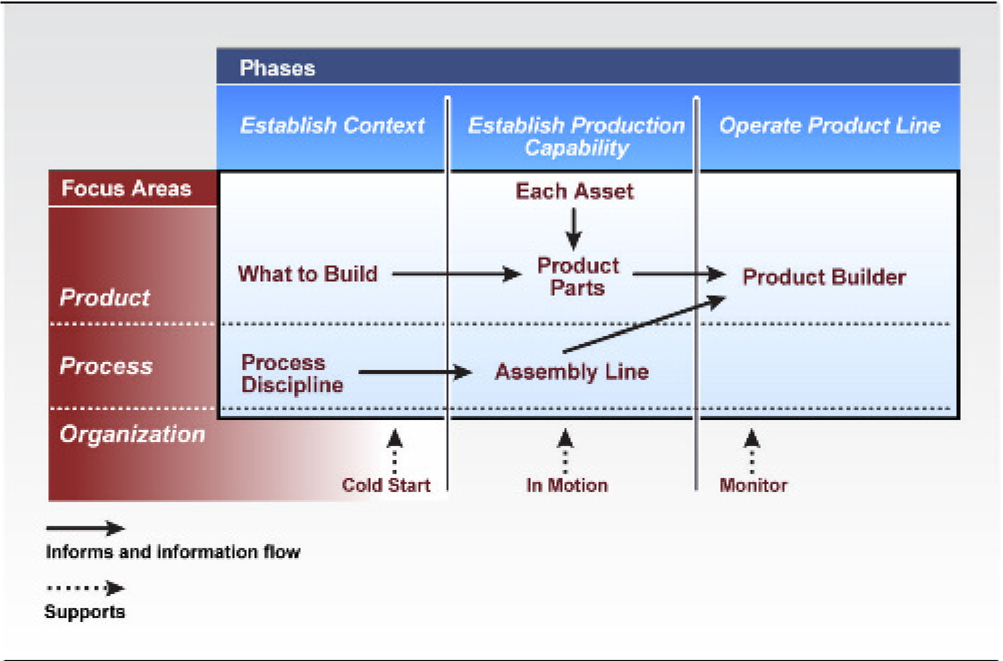


*Figure 15: Adoption Factory Pattern Annotated with Adoption Phases and Focus Areas*

11 "Process Discipline" is actually a practice area. This practice area is singled out because experience has shown that organizations that lack the ability to define and follow processes, even lightweight or agile ones, need to address that deficiency early in their adoption path.

Other useful views include the *practice area* view, which shows the constituent practice areas associated with the subpatterns, the *roles* view, which describes the principal roles involved in each of the nine cells, and the *outputs* view, which describes the key artifacts produced.

**Use the IDEAL model:** In the technology change domain of process improvement, the IDEAL model has enjoyed wide success [McFeeley 1996a]. With some generalizations, the IDEAL model is also appropriate for the launching of a product line. As shown Figure 16, the IDEAL model is iterative, allowing the reevaluation of the changing organizational context as the technology change project proceeds. This iteration also makes the model applicable to launching a product line effort from different levels of product line sophistication and to improving or institutionalizing the product line effort. This iterative cycle may be executed as many times as necessary to achieve the desired organizational state.

The model consists of five phases, each providing a basis for the next phase, with the final phase feeding back to the beginning. The five phases are Initiating, Diagnosing, Establishing, Acting, and Learning:

- An organization typically enters the Initiating phase as a result of some stimulus that intends to change the current way of doing business. In response to this stimulus, the appropriate sponsorship is established, and the appropriate resources are allocated.
- In the Diagnosing phase, the organization performs a diagnostic activity to baseline the current practices and probe for improvement opportunities.
- In the Establishing phase, the recommendations of the diagnostic activity are prioritized, change implementation teams are established, and plans are developed for conducting the activities.
- In the Acting phase, the planned activities are carried out.
- The end of a cycle is represented by the Learning phase. During it, the organization collects lessons learned that can then be applied to subsequent rounds of the technology change cycle.

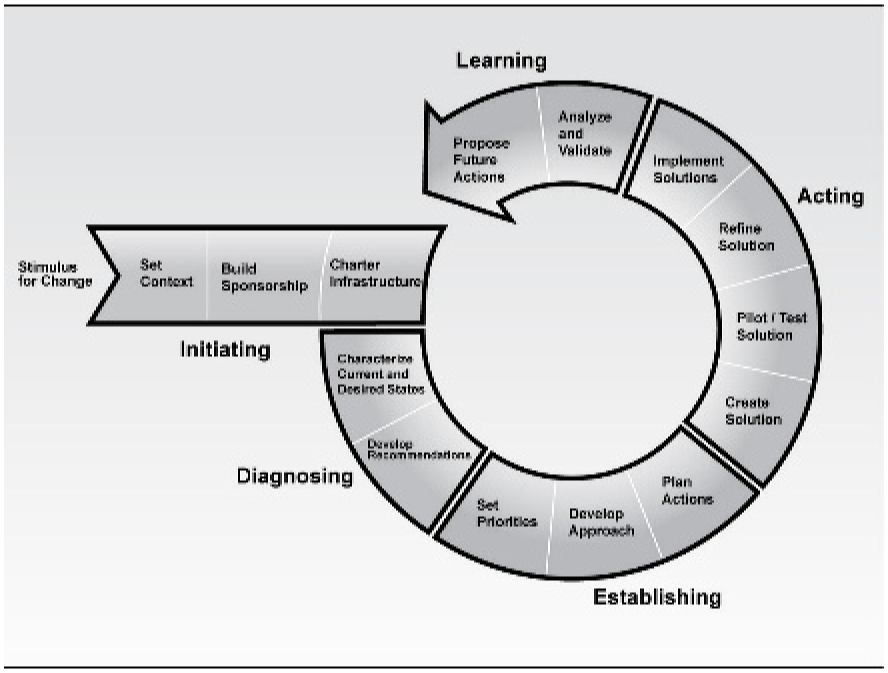


*Figure 16: The IDEAL Model for Technology Change*

Recognize that a given IDEAL iteration cycle may emphasize or de-emphasize a particular phase depending on the iterations that have gone before. For example, it is typical for more attention to be needed for the Initiating phase of early cycles than of later cycles. Specific details might be best determined by evaluating the risks associated with each phase, much as in the Spiral development model [Boehm 1988a].[12]

**Use the Adoption Factory pattern and the IDEAL model together:** As noted, the Adoption Factory pattern lays out the change that needs to occur when moving to a product line approach, but it lacks change management mechanisms and guidance about how to perform incremental adoption. On the other hand, the IDEAL model is a general model for guiding change but lacks product-line-specific guidance. These two models may be used effectively together when informed by contextual information about an organization (e.g., cultural aspects, degree of process discipline, and other particular strengths and weaknesses of the organization).

Below, we describe some ways in which the Adoption Factory pattern can be used within the phase sequencing of the IDEAL model. Note that not all these activities would necessarily occur in a single IDEAL cycle.

- **Initiating phase-forming commitment:** Once a product line opportunity has been identified and substantiated (perhaps with early forms of a business case or market analysis, a scaled-down use of the *What to Build* subpattern in the Adoption Factory pattern), promote management and staff

[12] Jones discusses how the IDEAL and Spiral models complement each other [Jones 1996b].

awareness of the opportunity, obtain staffing and resource commitments, and set product line objectives so as to meet specific business needs. The Adoption Factory pattern can serve as a common product line adoption vocabulary. The various views can be used to educate people on the overall approach, roles, and work products.

- **Diagnosing phase—assessing product line conditions:** In an early IDEAL cycle, evaluate the business and technical viability of a candidate product line opportunity (based on the "Scoping," "Building a Business Case," "Market Analysis," "Funding," "Organizational Risk Management," and "Technology Forecasting" practice areas). In a later IDEAL cycle, an organization could also use all the practice areas to determine areas for diagnosing organizational product line practices; this is the basis of the SEI Product Line Technical Probe (PLTP)[13] [Clements 2002c, Ch. 8]. The SEI uses the Adoption Factory pattern to organize the PLTP results that will serve as a gauge for how far along the organization is in its adoption effort.
- **Establishing phase—planning product line adoption:** Based on the results of the diagnosis, develop near-term and longer term goals and build an action plan to achieve them. The Adoption Factory pattern phases provide a guide for what might be reasonable goals; the *practice area* view provides specific technical targets. As a risk-reduction method (particularly in an early IDEAL cycle), this plan could include pilot projects to prove concepts and demonstrate their application within the organization. Such a plan will identify and schedule any actions (with the associated resources) that are needed to improve the organizational capabilities for undertaking a product line approach. In general, the performance of this practice should adhere to the principles of the "Technical Planning," "Organizational Planning," "Technical Risk Management," and "Organizational Risk Management" practice areas.
- **Acting phase—monitoring direction and performance:** Using the identified goals and reporting requirements from the plan created in the Establishing phase, monitor and evaluate the plan's outcomes and identify any needed revisions in your approach to accommodate changes in your objectives or opportunities for improvement. Activities in the acting phase should adhere to the principles set forth in the "Technical Planning" and "Organizational Planning" practice areas and be based on the results of data collection and metrics activities (see the "Measurement and Tracking" practice area).
- **Learning phase—tuning and improvement:** Risk management, planning, and measurement activities guided by the Adoption Factory pattern will identify any places where the product line effort is not yet a good fit with the organization's own special context. In this phase, the processes and organizational structures are modified to reflect lessons learned and take advantage of potential optimizations. Note that weaknesses or lightweight implementations in earlier Adoption Factory pattern phases that will need to be strengthened before the next IDEAL cycle.
- **Subsequent execution of the IDEAL cycles—promoting institutionalization:** Continue executing IDEAL cycles until all the practice areas in the Adoption Factory pattern are implemented with sufficient quality and rigor. Execute additional IDEAL cycles as appropriate to keep the product line up-to-date and responsive to changing conditions.

---

[13] Linda Northrop summarized the results of applications of the PLTP during an experience report at SPLC 2005.

**Use pilot projects:** Pilot projects can be an important means of reducing risk and learning more about the organizational and technical issues associated with the product line effort. A successful pilot project can also be an effective means of building advocacy. A pilot should be viewed as a controlled experiment to test specific ideas or concerns.

Some criteria to consider when selecting a pilot include its

- **scope:** The pilot should be scoped such that it can be executed in a relatively short time frame with relatively limited resources.
- **importance and visibility:** The organization should care whether the pilot is successful. In other words, a success in the "backwater" of the organization is unlikely to generate any enthusiasm. On the other hand, the pilot should not be so important—for example, on a critical path—such that failure has a significant adverse impact on the organization.
- **probability of success:** Especially for initial pilots, the effort should have a reasonable chance to succeed. If an organization already has success with product lines, more risky pilots might be chosen to test specific concepts.
- **choice of participants:** Unless the pilot is aimed specifically at testing organizational resistance, the participants in the pilot should be advocates (or at least be open-minded).

**Use proactive, reactive, and incremental approaches:** As noted in "All Three Together," organizations may choose to take a *proactive, reactive, or incremental* approach to product line adoption. As discussed, each approach has its advantages and disadvantages.

**Use lightweight approaches at first:** When initiating a product line effort, some organizations report success with using lightweight organizational structures and lightweight processes to support them. These lightweight approaches facilitate smoother transitions to product lines that can be changed quickly and nimbly as the organization learns what does and does not work for its particular situation. It is worth noting that this approach is a form of piloting. Clements and Krueger debate the advantages and disadvantages of lightweight and other approaches [Clements 2002b].

**Use product line diagnostics:** Before an organization can launch and institutionalize a product line capability successfully, it needs to know its blind spots. If it is lacking necessary expertise in one of the practice areas (especially one that tends to manifest itself early in the product line life cycle, such as scoping or requirements engineering), it is unlikely to make a successful (let alone smooth) transition to sound product line practice. However, by recognizing potential trouble spots early, you can focus judiciously on resources and shore up any areas of weakness that, if left untreated, would undermine the product line effort.

One instrument to help an organization identify problem areas is the SEI Product Line Technical Probe (PLTP) [Clements 2002c, Ch. 8]. This diagnostic identifies an organization's strengths and challenges in each of the product line practice areas. These results provide the grist for an adoption or launching plan, the first part of which will be to rectify any deficiencies in critical skill areas. Other product line diagnostics include the SEI Product Line Quick Look (PLQL), the Bosch Product Line

Potential Analysis [Fritsch 2004a] and the European Union Information Technology for European Advancement (ITEA) Business, Architecture, Process, Organization (BAPO) evaluation [van der Linden 2004a].

**Develop product line goals, objectives, and strategies:** Technology change should be initiated not for its own sake but to support specific organizational goals. Thus, an early step in the technology change project is to establish an appropriate set of goals, which are validated by a supporting rationale. A set of objectives should also be determined to provide high-level, measurable indicators of progress toward the goals. Given a set of goals and supporting objectives, a number of different strategies are typically appropriate to the achievement of those goals. Presumably, one of those strategies is to initiate (or expand) a product line effort. Another strategy might be to build (or further mature) a software process infrastructure to provide a foundation for the product line effort. An internal workshop is an excellent vehicle for articulating and capturing the goals, objectives, and strategies that will serve as the foundation for building an adoption plan. As product line adoption proceeds, the organization should revisit those items and update them as necessary.

**Use process improvement as a basis for launching and institutionalizing:** Process discipline provides a foundation for product line practice. As indicated by the "Process Discipline" practice area's placement in the Establishing Context phase of the Adoption Factory pattern, it is likely to be in your organization's best interests to initiate a process improvement effort early in, if not prior to, your software product line adoption.

Many organizations use SEI Capability Maturity Model Integration (CMMI) for Development with Integrated Product and Process Development (IPPD) as the basis for their process improvement efforts [SEI 2006a]. While some of the process areas on the CMMI models provide a basis, there is always something extra required to transform a single-system-oriented process to an appropriate corresponding product line practice.

CMMI is certainly not the only model for process improvement; many organizations have adopted a Six Sigma approach [Brassard 2001a]. Originally developed in the 1980s by Motorola to address electronics manufacturing quality, Six Sigma has evolved into a philosophy, an improvement framework supported by a set of improvement tools, and a structured approach for business improvement. The philosophy of Six Sigma is to improve customer satisfaction by reducing and eliminating defects, resulting in greater profits. The Define-Measure-Analyze-Improve-Control (DMAIC) Framework is a means to improve existing processes and products.

With its "Plan, Do, Check, Act" roots, DMAIC has been described as a "weakness-based," tactical approach to getting to the bottom of a problem through quantitative means.

## Practice Risks

The risks of launching a product line relate to misapplying the product line approach by either failing to institute beneficial practices or instituting practices that are not appropriate to the particular organizational situation. A poor launching and institutionalizing strategy will result in failure of the product line to meet its goals, very likely because the staff will fail to accept it as a way of doing business. Such a failure can result from

- **lack of an identifiable champion:** Any institutional change requires a strong champion who can effectively communicate the vision, console the troops when things go badly, make sure that the organization is following the new ideas, remind everyone why the organization has decided to embrace the change, and never, ever let enthusiasm wane. A software product line needs such a champion who has the management authority (or has the ear of management authority) to keep the organization on track. An organization lacking an effective champion will have a much more difficult time adopting the product line approach.
- **approach mismatch:** If the products being developed do not have sufficient commonality to warrant a product line approach, any launching effort will fail. Software product lines should help your organization meet specific business goals. In other words, a product line approach to software is a means to achieve an end, not the end itself.
- **inadequate management commitment:** If management has not been convinced of the viability of a product line approach for their situation, the funding, staffing, and other resources may be inadequate for a successful adoption. Mitigation involves developing additional evidence of the opportunity (perhaps through additional market analyses or narrowly focused pilot efforts) that is aimed at addressing specific concerns. Additionally, efforts may focus on developing the commitment at high organizational levels at which there may be more strategic perspectives on the business situation and future needs.
- **insufficient staff commitment:** If the technical staff has not been convinced of the viability of a product line approach, they will be likely to torpedo the effort. A major aspect of any product line launch should focus on educating the staff and achieving their buy-in. Involving technical staff in the definition of plans and processes is one mitigation step.
- **insufficient bounding:** If the planned effort requires too much effort over too long a period of time, the attainable benefits may not be realized soon enough to justify the investment. Mitigation might involve limiting the product line scope to create high-payoff core assets or limiting the market focus to address critical or selected work products so that this effort can concentrate on delivering valuable benefits within the needed time frame.
- **inappropriate application:** If an organization does not have a clear product line charter or has one that is different than its stated charter, it will probably build products that are too divergent from the planned product line focus to benefit as anticipated from the product line approach. Mitigation involves ensuring that knowledgeable and experienced management, engineering, and marketing staff are involved over an adequate, but not excessive, period of time in resolving the true product line charter to be pursued. The organization should try to ensure that the needed diversity among future products, which must often be avoided with conventional development approaches, is accommodated properly.
- **premature standardization:** If institutionalization occurs too quickly, the organization may institute inappropriate standards and terminate innovation prematurely.
- **missed or delayed standardization opportunities:** On the other hand, if standardization opportunities are missed or delayed, there may be redundant or unnecessarily divergent efforts and less than optimally effective practices.

- **insufficient tailoring:** If standardized practices are tailored insufficiently or inappropriately, the organization will probably adhere to suboptimal practices or unsupported deviations from the preferred practices.
- **failure to evolve the approach:** If the approach to software product lines is not improved continuously over time, practices will probably become ineffective, and unsupported deviations will crop up out of necessity.
- **ineffective dissemination:** If there is an inappropriate level or type of documentation, inadequate training, or ineffective support, the product line launch will most likely not take effect in the planned form or produce the desired outcomes in the required time frame.
- **lack of an emerging community:** A champion is essential to launch a product line effort, but a champion cannot institutionalize product line practices. It takes a community of individuals who are from various parts of the organization and who are committed to the product line approach to ensure institutionalization. If a community is never formed, the product line practices will erode over time.

## Further Reading

[Ardis 2000a]
Ardis and colleagues describe the steps that product line advocates took at Lucent.

[Boeckle 2002a]
Boeckle and colleagues recommend an approach for adopting and institutionalizing a successful culture of product lines in an organization.

[Bosch 2002a]
Bosch proposes a set of bellwether indicators of an organization’s sophistication with respect to software product lines. These indicators can be used to set adoption ambitions appropriately and help produce a feasible launch strategy.

[Brassard 2001a]
Brassard and Ritter provide an easy-to-read, quick reference to the “art and science” of Six Sigma.

[Clements 2002c, Ch. 7]
Clements and Northrop describe 12 basic product line practice patterns that help effect a “divide and conquer” approach to launching and institutionalization.

[Clements 2002c, Ch. 8]
Clements and Northrop describe the Product Line Technical Probe in detail.

[Fritsch 2004a]
Fritsch and Hahn describe the Bosch product line diagnostics.

[Northrop 2004a]
Northrop describes the Adoption Factory pattern and its views and use in detail.

[Wappler 2000a]
Wappler from IBM Consulting Group focuses on a pilot project as a way to mitigate risks during launching.

[van der Linden 2004a]
Van der Linden and colleagues describe the BAPO product line diagnostics.

## Market Analysis

Market analysis is the systematic research and analysis of the external factors that determine the success of a product in the marketplace. It involves the gathering of business intelligence, competitive studies and assessments, market segmentation, customer plans and strategies, and the integration of this information into a cohesive business strategy and plan. For the purpose of this practice area, we define *market* as the place where people meet for the purpose of trade. Those involved include sellers, buyers, prospective customers, providers of complementary products or services, competitors, and any other party that "participates" in the day-to-day conduct of business transactions.

Organizations conduct market analysis as a means of characterizing quantitatively the business opportunity for their products. From this analysis flows the statement of organizational objectives for cost, quality, and productivity and any associated constraints. Based on this analysis, a business case, strategy, and plan can be developed. The goal of market analysis is to provide sufficient detail to answer the fundamental question: "Does the market represent sufficient economic potential for us to achieve our business goals with this product?"

Even defense organizations considering the procurement of a military system must ask themselves a version of this question. While some commercial market analyses may slip in colorful military metaphors (such as "doing battle with the competition"), these terms are literal for defense groups. For them, a market analysis becomes a "mission" analysis; the competition is referred to as the "threat" or (less obliquely) "the enemy." Time to market becomes time to deployment, time to field, or time to operational readiness. Competing systems are the "counterforce." And whereas the return on investment (ROI) is not the overriding business goal, effectiveness and protecting the war fighter are. Such a market analysis may look very different from one produced in a software start-up company looking for venture capital, but the need for it is the same: to answer the question "Will this product be successful?"

Not surprisingly, market analysis provides a fundamental input to the business case for a product.

Because markets change and evolve, a market analysis should be a living document that guides the decision making throughout the life of the product or products included in the product line's scope. Many organizations couple their ongoing market analysis with an annual planning and budgeting cycle, so that product development and evolution priorities are explicit and integrated into the plan.

### Aspects Peculiar to Product Lines

A market analysis is an important ingredient in a decision to shift an organization's business strategy from single-system engineering to product line engineering. The market analysis informs the business case, and together these two documents form the basis of the decision package used by management to justify the shift in strategy. Once a product line engineering organization has been established, a market analysis is conducted on a continuous basis to guide the introduction of new products into the product line, to steer the evolution of the product line as a whole, and to inform the spin-off of related product lines.

The “Building a Business Case” practice area relies on business assessments of forecasts, market share, and pricing. It highlights the relationship between the market analysis and the organizational need for the business case. That is as true for the ongoing product line operation as it is for the organization making the decision to shift strategy to a product line operation.

In a product line organization, responsibility for the market analysis lies within a strategy formulation or business development function or with the individual responsible for the success of the business unit or product line—whether it be a general manager, product line manager, or product manager.

While engineering and research and development (R&D) personnel will certainly contribute, they would not normally take responsibility for market analysis. In this way, organizations avoid building products for which there is no market or building elegant solutions before searching for a problem. In a similar way, the sales organization is a significant, but often optimistic, contributor to the analysis because it knows the market. A sales contribution might provide the answer to some key questions, such as

- “Given a set of products with *these* rough distinguishing capabilities available around *these* dates with a projected price targeted for about *these* amounts, how many of each product in the product line can you sell in Year 1? Year 2? Year 3?”
- “Can you commit to this forecast, and do you agree to be held accountable for its achievement? If not, what level of specificity is required for the capabilities, dates, and amounts so that you will agree to such accountability?”

Answering these questions leads the market analyst to the answers to other questions, such as

- “What customers or market segments will purchase which products?” Or, less concretely, “Which features or feature combinations are important for which customers?”
- “What are the similarities and differences in their buying patterns?”
- “What features are available and how are they (or how might they be) configured for each product within the family?”
- “Who else offers products that overlap with our product line?”
- “What trends are observable in the market that may affect the answers to any of these questions?”

Armed with quantitative business goals and constraints, a characterization of the market requirement, and known engineering competencies, an organization can make informed decisions concerning the viability of introducing a product line or new products in a product line.

Market analysis helps determine the feature-binding time by defining the feature delivery method [Kang 2002a]. During core asset development, those features that are common to all products are bound into the core assets. Those features specific to a product are provided for during product development and may be bound then or at later times such as product configuration time.

### Application to Core Asset Development

Market analysis is one of the early steps that help determine the product line scope—that is, what products the product line will comprise, suggesting a first-order approximation of the product line's commonality and variability. The scope, of course, leads to the architecture, which leads to the software core assets. The result of a market analysis helps provide a customer and product profile around which a focused R&D or engineering program can be designed. The organization makes core asset development or acquisition investment decisions within the context of customer or market requirements for the product, thus giving the engineering program a goal.

Beyond this, however, the market analysis is, itself, a core asset—one that is created when the product line is launched but maintained and updated as the market changes, as new members of the product line are considered, as the product line itself evolves, and perhaps as a new product line is spun off. In the last case, an organization can use market analysis to identify untapped market areas that it can exploit through a product line's existing core asset base.

Market analysis also provides input to the development of the production strategy. One approach to developing the production strategy for a software product line is to resolve the five forces identified in Porter's strategy development framework [Porter 1980a]. In this approach, the organization examines the forces exerted by competitors currently in the targeted market and potential entrants into the market. Then, it develops business and product production strategies to counter these forces. For example, the product line organization may determine that price is a critical issue in the market and decide to design the core assets so that products can be generated automatically. That lowers their manufacturing costs, thereby allowing price reductions that make the market less attractive to potential entrants.

### Application to Product Development

The marketplace is an ever-changing environment, so market analysis needs to be an ongoing activity that continues through product development. Market analysis helps the product developers factor other customer preferences into the product definition. The knowledge of customer preferences drives the decisions about features and feature combinations, as well as quality attributes such as performance, availability, and configurability.

A more detailed knowledge and understanding of the needs of specific customers or customer groups lead to decisions about the number of discrete products within the product line and how these discrete products are defined. Among specific products, there are choices of configuration. What features are selectable or definable by the customer? What is the range of options available for each product within the product line?

### Example Practices

A market analysis can be conducted by executing the steps outlined below.

**Identify information sources:** The market analysis starts with the identification and location of any information resources that could have an impact or provide insight into the definition of the product line. In identifying these resources, it is important to consider some basic attributes of the resource or information being provided, such as: Is it accessible and reliable? Will it reveal valid needs or simply a passing fad? Is it relevant to the product line at hand? Sources of information include sales calls, meetings, conferences, customer services, newspapers and magazines, past performance, focus groups, surveys, consultants, budgets, published planning data, economic indicators, technology trends, and other intelligence sources.

**Gather information:** The next step is to gather the information necessary to get a broad market definition, with additional details about specific segments that most closely align with the products within the product line. The analyst should assume that every contact with every customer, potential customer, user, competitor, or other market participant should be exploited for the information it can produce: requirements, a list of features liked/disliked, pricing, competitive trends, business goals and strategies, purchasing plans, budgets, and so forth. This broad market survey provides the product line organization with detailed knowledge and an understanding of the market, their potential customers, and the other market players.

**Identify customer segments:** Next, the analyst focuses on the similarities and differences among the product usage characteristics of specific customers. This study makes distinctions among customers evident and helps the analyst make and categorize generalizations about different classes of customers. The analysis shows preferences for such things as price, configuration, availability, and technical approach. Other environmental concerns, on a product-by-product basis, results from the analysis of these segments: economic potential, the strength of the competition, standards, the rate and direction of technological change, and other key factors critical to the customer’s purchasing decision.

**Map products to segments:** In light of this market information, the analyst can compare the capabilities of each product or proposed product within the product line to the customer’s expectations and requirements within each segment. Based on the degree of alignment between the products in the product line and the segments the product line was designed to satisfy, the analyst can make recommendations concerning the scope, magnitude, and direction of the organization’s core asset, product development, and acquisition activities.

**Examine the competition:** The analyst can compare individual products in the product line with those offered by competitors. The purpose of this comparison is to evaluate the product line’s strengths, weaknesses, and competitive differentiators. Based on the outcome of this analysis, the organization can make recommendations concerning product development investments as well as product positioning relative to important competitors.

## Practice Risks

Without an adequate, thorough market analysis, the wrong products are likely to be built—wrong in their overall configurations and in how they respond to customer or market requirements. An inadequate market analysis can result from the following factors:

- **an inaccurate forecast of the market size:** If the forecast of the market size is inaccurate, the product line organization is likely to be unable to achieve a sufficient ROI from product line development. That could result from an inadequate analysis of the economic potential of the segment or from the organization's inability to penetrate the segment as a result of product weaknesses, unanticipated competitive strengths, or other compelling factors. When this situation occurs, the product organization must face the decision to pull the plug or to increase or redirect investments in the project.
- **right product, wrong market:** Analyzing the wrong market—for example, by interviewing the wrong market groups—will result in an analysis that appears valid but, in reality, does not apply.
- **right product, wrong price:** Having the right product but approaching the market with the wrong price, partners, suppliers, or sales channel can also be a risk. These factors can influence a product line's success or failure as much as the product itself can.

### Further Reading

[Faulk 2000a]
Faulk, Harmon, and Raffo provide a tool for analyzing the value of a product as perceived by a customer.

[Kang 2002a]
Kang, Lee, and Donohoe discuss the role of market analysis in determining feature-tree binding.

[Porter 1980a]
Porter's Competitive Strategy is required reading for anyone responsible for conducting market analyses.

[Prahalad 1990a]
Prahalad and Hamel make clear the imperative of developing organizational capability to build products that align with market requirements.

## Operations

This practice area is about how business gets done; it is the engine that makes the organization run. Without it, the organization is mostly just a collection of staffed units, poised and willing to do the right thing but unsure of precisely what that is. Any organization in the business of developing products operates under the aegis of its organizational and/or management strategies and policies, business processes, and work plans. Operations puts all of these items together into a coherent unified corpus of policies and practices. The vehicle for documenting these policies and practices is an *operational concept* [ANSI 1992a]. The operational concept

- describes the processes for fielding and maintaining the products from an operational perspective
- describes how the organizational units work together to execute these processes
- defines the role that acquisition will play and points to defined acquisition strategies and policies
- facilitates a common understanding among members of the organization as to how products are fielded and how the production capability is evolved and maintained

- serves as a baseline when the organization considers alternatives in its approach as warranted by changing conditions

### Aspects Peculiar to Product Lines

With product lines, this practice area spells out how the organizational structure populates and nurtures the core asset base and how it uses the core asset base to build products. In the figure that shows the product line's essential activities, operations tells how the wheels are set in motion, how the right information flows along the arrows, and how management orchestrates the entire business.

Operations breathes life into the organizational structure that was put into place to carry out product line activities. The "Structuring the Organization" practice area positions people in the right work units and assigns them broad responsibilities. However, without a clear definition of the operational concepts for building and running the product line, those people will flounder or, at best, perform inefficiently. To borrow an assembly line analogy, structuring the organization is like telling automobile workers, "Stand right here, and here, and here. When parts come along, put them together so that an automobile comes out at the end." That's not enough instruction. Operations, on the other hand, tells the workers how the pieces fit together and in what order, what to do when the right pieces aren't in place or don't fit together, how to report pieces of poor quality, how to make suggestions for improvement, how to interact with workers standing at stations near theirs, and what they are and are not allowed to do (for example, "Don't bend the pieces to make them fit together!"). Both structure and operations are necessary for the assembly line to roll out quality products.

The product line approach is not a case of "one size fits all." Operations spells out how the approach is implemented in organization-specific terms.

Operations describes how and which organizational units

- produce and evolve core assets according to those assets' work plans and evolution plans
- define and evolve the production plan
- use the core assets and production plan to field products
- continuously monitor and improve the health and profitability of the product line

Operations also describes the interconnections among those carrying out the essential activities of core asset development, product development, and management. These interconnections would include communication mechanisms, decision and conflict resolution processes, a document map, and a supporting Web site or wiki for the product line.

The best way to document the operational concept is to develop a formal product line concept of operations (CONOPS). An organization develops this document (which is, itself, a core asset) to describe its product line approach in organization-specific terms. The CONOPS documents the decisions that define the approach and the organizational structure needed to put it into operation.

The more conventional purpose of a CONOPS is to represent the user's operational view for a system under development. This view is stated in terms of how a system will operate in its intended environment. The CONOPS for a product line serves much the same purpose but for the product line organization. The CONOPS documents the organization's decisions about its product line operations—that is, how the members of the organization work together to accomplish core asset development and product development and manage and orchestrate the product line effort. All product line personnel use the CONOPS as the guide to how to carry out any product line action or communication. In particular, managers use the CONOPS to determine who is affected by a decision and who needs to be informed about specific decisions. Technical members of the organization use the CONOPS to determine who to contact for decisions.

Someone, usually the product line manager or a Product Line Steering Group, needs to own the operational concept, make sure it is well documented, effectively communicated, followed in everyday business, and improved as necessary based on measurement and feedback from individuals within the organization and customers of the product line.

### Application to Core Asset Development

For core asset development, operations defines what plans, processes, strategies, policies, and constraints the core asset developers will carry out to do their jobs and how those items will be managed. Therefore, the operational concept for core asset development documents the organization's decisions regarding

- the organizational elements and the role each of them plays relative to the core assets
- the production strategy to be used for core assets and the production strategy to be used for products
- the core asset development activities moving from scoping through requirements engineering, architecture definition, component development, using existing available software, the mining of assets, testing, and software integration
- the strategy for maintaining the core asset base, which covers the process for creating new core assets, modifying existing core assets (including changing the architecture), and declaring core assets obsolete
- specific guidance for supporting the shared responsibilities of core asset evolution. Such guidance guarantees that the architecture and assets will be used to produce products and that feedback will be sent about that use to those responsible for core assets. That feedback helps to continuously improve those assets.
- communication mechanisms used to coordinate core asset development activities with product development, and with configuration management, measurement, and other management activities

### Application to Product Development

For product development, operations defines what plans, processes, strategies, policies, and constraints the product developers will follow to do their jobs and how those items will be managed. Therefore, the operational concept for product development documents the organization's decisions regarding

- the organizational elements and the role each of them plays in product development

- how the organizational elements effect the production strategy and use the production plan to carry out the product development activities
- the strategy for maintaining the products and coordinating that maintenance with the evolution of the core assets, including configuration management policies and processes
- specific guidance for feeding the results of using core assets in product development back to those responsible for core assets in order to support the continued improvement of those assets
- specific guidance for recommending new core assets or changes to existing core assets

Finally, the production strategy for products, described in "Core Asset Development," is reflected in the operational concept.

## Example Practices

**Creating a CONOPS:** The stakeholders in the product line organization need to participate in the creation of the CONOPS in order to ensure that the operating concept is realistic and to mitigate the risks associated with failure to buy into the product line approach. Holding a workshop for the formulation, or at least the review, of the CONOPS is one way to involve stakeholders in its definition.

**Managing the operations:** A product line manager, or an equivalent organizational unit that performs the role of product line manager, orchestrates the product line effort. Suggested managerial practices for the product line manager include

- defining and articulating the product line vision and promulgating it throughout the organization consistently and often—in fact, every day. That can be done by posting the vision in a publicly accessible location (such as an intranet Web site), crafting slide presentations to be given by organizational and technical managers, and discussing the vision at opportune times, from brown-bag lunches to management seminars.
- creating and posting both personal and organizational performance objectives that embrace the product line goals and strategies
- establishing promotion and reward structures that provide real benefits to individuals who follow the documented product line approach to building products, contribute to the improvement of the product line effort, and design and build core assets that are, in fact, reusable
- communicating product line progress early, openly, and often
- removing from the critical path (or from the organization) those individuals who are barriers to product line success because of either a lack of productivity or a consistent lack of alignment with defined product line practices
- learning about technology change management, taking formal training if possible
- championing the product line at higher levels of management
- protecting the product line staff from unnecessary distractions (perhaps imposed by higher management or requested by overfamiliar customers) that do not further the product line effort
- ensuring stable funding for true "operation" of the product line. The funds must cover not only the development and evolution of the core assets but also the entire supporting infrastructure.

- integrating efforts across organizational boundaries by relying on support and assets from other parts of the organization or other organizations

**Developing scenarios:** Scenarios can help those developing a product line CONOPS gain an understanding of product line operations. You can create scenarios for the following tasks:

- using the product line architecture and other core assets to build a product
- developing product-specific assets
- submitting new or modified components to the core asset base
- updating the core asset base and migrating the changes into existing or in-development products
- determining whether a candidate product is in or out of the product line scope
- delivering product line systems to customers
- supporting the implementation and maintenance of the development and execution environments for product line systems

An organization can cast these scenarios as a series of “A Day in the Life of . . .” vignettes. Because they are focused and lack extraneous information, these vignettes make product line activities come alive for individual or small groups of role players. The role players themselves could be asked to draft the vignettes, thereby testing their understanding and encouraging more enthusiasm and buy-in than scenarios developed by management would.

## Practice Risks

An inadequate or inappropriate operational concept will result in the product line staff working at cross-purposes and in conflict with each other. Productivity (and morale) will drop. Also, (depending on the nature of the problem) core assets might not be used in the way they were intended, products might not be turned out as planned, and the product line might stagnate and die. An inadequate operational concept can result from the

- **lack of management commitment and/or leadership:** Product line operations, most especially during the early stages of a product line effort, require leadership, commitment, and follow-through. Setting in place an effective product line operational concept often requires culture changes and changes in the organizational structure.
- **lack of the necessary ingredients:** Product line operations rely on the existence of plans, processes, strategies, and decisions relative to organizational structure, lines of authority, communication, measurement, and so forth. Without these necessary ingredients, the operations will falter, the core asset and product wheels will not spin, and the information will not flow.
- **failure to identify a product line management role:** Success of the product line requires strong visionary management. Some specifically designated manager or group of managers must maintain the vision and keep the organization aligned with that vision during the period of change. In particular, the person or entity acting as product line manager needs to oversee the development of the product line operational concept personally and obtain the buy-in of key stakeholders.
- **failure to update the operational concept:** Operations will evolve and so must its documentation. The operational concept must not become shelfware if the organizational engine is to keep

churning efficiently. The operations and its documentation should be reviewed and revised constantly as the product line is fielded, lessons are learned, and the product line evolves. If the operational concept is not maintained, newcomers will not have sufficient orientation, and new managers will have a tendency to undo what has become effective. Moreover, the product line may not evolve successfully to address the needs of new customers.

### Further Reading

[ANSI 1992a]
The American National Standards Institute (ANSI) provides a standard for preparing operational concept documents.

[Cohen 1999a]
Cohen applies the idea of using a standard to product lines and introduces a generic product line CONOPS that can be suitably tailored and adapted by an organization to meet its specific needs and circumstances. Cohen's report includes guidelines and scenarios to help an organization that has proposed a product line, provides a nearly complete example of a CONOPS for a product line approach, and details the class of information to be contained in each section of a CONOPS.

[IEEE 1998b] IEEE describes the format and contents of a CONOPS.

[McGregor 2005a]
As part of the SEI Pedagogical Product Line, McGregor provides a sample product line CONOPS for a hypothetical company. This document gives a useful outline for a CONOPS and example contents.

## Organizational Planning

Organizational planning pertains to strategic or organizational-level planning. (For information on the foundations of planning, see the "Technical Planning" practice area.) Organizational planning relies on these same foundations, but its scope transcends individual projects.

As discussed in the "Technical Planning" practice area, when considering the planning activity, it is useful to distinguish between the *process* by which plans are created and the *plans* that result from that process. The process for generating plans is often very similar regardless of the organizational level at which it is applied. The process should differ primarily by who is involved and the scope of the effort to be planned. For organizational planning, senior and mid-level managers are often primary participants. The scope of the organizational planning process should include planning for cross-project activities or activities that are outside the scope of any project.

Regarding the plans themselves, there are different types of plans for addressing different purposes. Examples of organizational management plans include organizational strategic plans, funding plans, technology adoption plans, and organizational risk management plans.

As discussed in the "Technical Planning" practice area, a collection of interrelated plans is often more appropriate for accomplishing larger tasks than a monolithic master plan. Because of the broad scope of organizational planning, you should expect dependencies to exist among these plans and subordinate project plans. These relationships should be an explicit part of the plans.

### Aspects Peculiar to Product Lines

There is nothing fundamentally different about a *planning* process for product lines. However, there are certain types of organizational management plans that are unique to product lines including

- **product line adoption plans:** These plans describe how to transition an organization from its current way of development to a product line approach. (See the “Launching and Institutionalizing” practice area.)
- **core asset funding plans:** Funding the development and maintenance of the core asset base is likely to be done at the organizational level. (See the “Funding” practice area.)

Besides these plans, other organizational plans that you might find in any development organization will take on a decidedly product line flavor. Organizational planning is used to facilitate the implementation of any of the technical management or organizational management practice areas that have organizational implementations [Clements 2005a]. The major plans are associated with

- **configuration management (CM):** In a product line effort, CM is more complicated and reaches across all of the core asset and product-building projects and possibly even across product lines. It is usually appropriate to plan CM at the organizational level.
- **tool support:** Tools are often considered core assets, and one of the savings a product line organization enjoys comes from using the same tool environment across all of its product efforts. If common tool support is provided and maintained across products in an organization, it should be planned organizationally.
- **training:** Like tool support and CM, it pays to consider training at the cross-product level. An organizational plan for training should address: identifying training needs, establishing and maintaining the training capability, providing the training, and managing the training process including record keeping and effectiveness assessment [SEI 2006a].
- **structuring the organization:** This plan will detail the transition steps and shifts in responsibility, outline any logistical or physical relocation, and assign schedules and resources.
- **risk management:** This plan will assign people to participate in the process, account for any training or other preparation required, and lay out an engagement schedule.

There may be a recurring need to develop some of these types of plans. For example, in the case of a phased rollout in different parts of the organization, it may be useful to develop tailorable plan templates or provide examples that serve as core assets themselves to seed the planning process.

In addition to the plan dependencies that result from being at the organizational level, there will be additional dependencies owing to the product line approach. Organizational plans will have dependencies with project plans, and project plans will have external dependencies among other project plans. Organization-level plans may be necessary to coordinate project-to-project dependencies. In a product line context, the project plans can relate to core asset development, product development, or activities that cross between them.

### Application to Core Asset Development

Organizational plans strongly related to core assets include those for

- **funding:** Typically, core asset funding issues must be addressed at the organizational level.
- **priorities for core asset development:** Especially when new core assets are being developed, product development projects may have competing needs for when particular core assets become available. Organization-level planning may be necessary to resolve the conflicts.
- **configuration management:** In a multiple-project product line effort, it may be appropriate to plan the CM of core assets at the organization level.
- **risk management:** Typically, an organizational risk management process would include risks related to core assets. Mitigation plans might be needed to address some of those risks.
- **product line adoption:** A significant part of a product line adoption plan will be a description of how core assets will be created and maintained initially.

And, as is the case with technical (or project) plans, the organizational plans themselves (or parts of the plans) make fine core assets. Ideally, reusable plans should be tailorable in the same fashion as other core assets—that is, they have defined points of commonality and variability. Cost, effort, and schedule estimates may be useful candidates, particularly for reuse, as are work breakdown structures, goals, strategies, and objectives.

### Application to Product Development

Typically, product development planning is handled below the organization level. Organizational planning would primarily provide constraints and priorities to guide project planning. Specifically, that might include

- **product line adoption:** The product line adoption plan should describe what initial product development will be accomplished (for example, which project or projects will serve as pilot efforts for launching the product line). Project-level plans would detail how that would be accomplished.
- **risk management:** Typically, an organizational risk management process would include risks related to product development. Mitigation plans might be needed to address some of those risks.
- **CM:** In a multiple-project product line effort, it may be appropriate to plan the overall CM at the organization level. Project plans would have to be compatible with the CM plan.

### Example Practices

Some characteristics of a good planning process and good plans are described in the “Technical Planning” practice area. These characteristics are equally applicable to organizational planning.

### Practice Risks

The primary risk is that the organization may fail to identify and effectively plan the activities that require organization-level planning, resulting in a muddled product line effort that will fail to meet its goals and expectations. Other risks cited in the “Technical Planning” practice area are also relevant to organizational planning.

**Further Reading**

[Clements 2005a]
Clements, Jones, McGregor, and Northrop discuss project management in a software product line organization.

[SEI 2006a]
CMMI for Development, V1.2 provides guidance about project planning.

## Organizational Risk Management

Organizational risk management is risk management at the strategic level. Risk management concepts are discussed in the "Technical Risk Management" practice area, which describes the activities that are necessary for project-level risk management. An organizational risk management process relies on the existence of such project-level risk management and provides mechanisms for surfacing and managing risks that transcend, or are shared across, projects.

The seven principles of risk management are shown in Figure 17. These principles are divided into one core principle, three sustaining principles, and three defining principles. An effective risk management program exhibits characteristics of all seven principles.

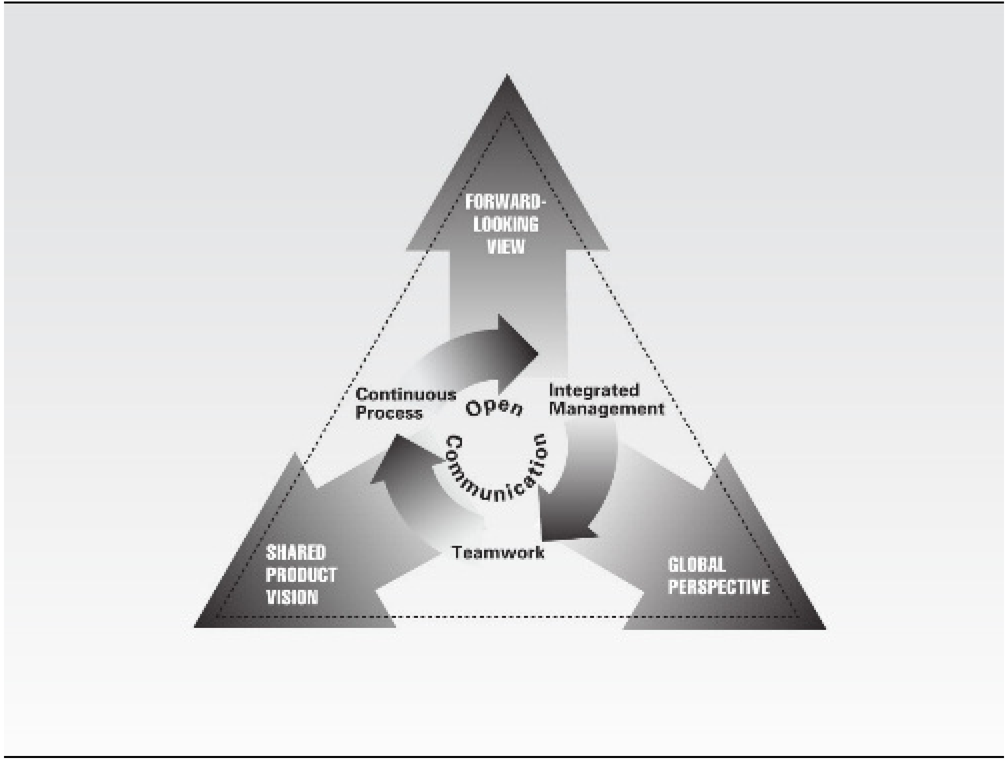


*Figure 17: The Seven Principles of Risk Management*

### Core Principle

- **open communications:** An effective risk management process must encourage the free flow of information at and between all project levels. The process should enable formal, informal, and impromptu communication.

### Sustaining Principles

Three sustaining principles allow an active risk management process to sustain its success in an organization:

- **integrated management:** When risk management is treated as a “bolt-on” or side activity, team members become frustrated, because managing risk is seen as interfering with their “real work.” Teams performing risk management must integrate risk identification, risk analysis, and risk planning tracking and control into normal product line management activities and forums. As depicted in Figure 18, risk management is integral to project management.

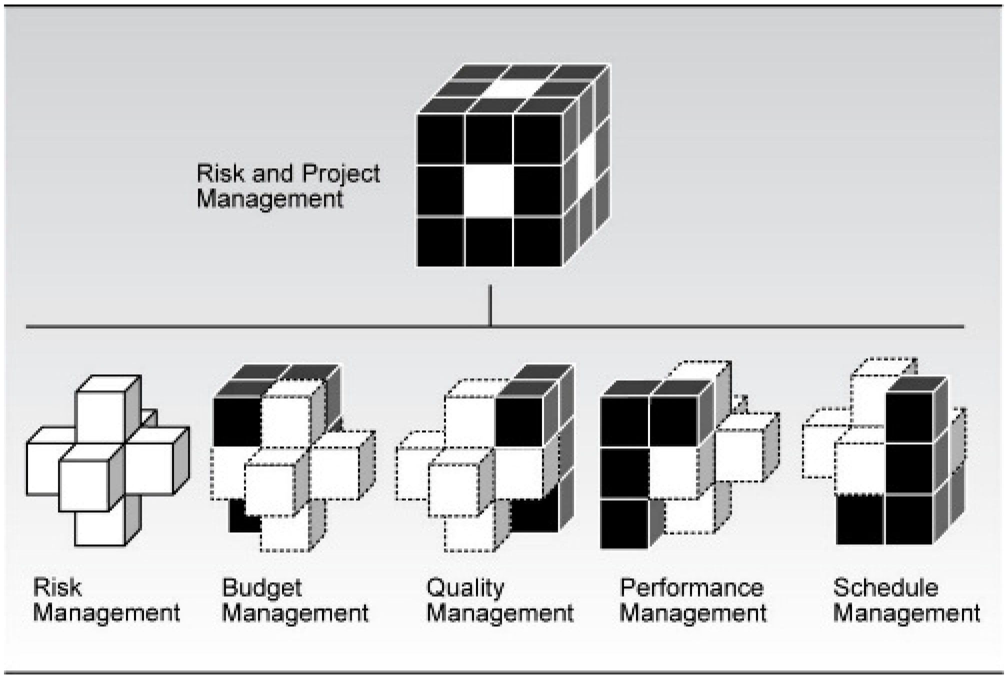


*Figure 18: Risk and Project Management*

- **teamwork:** The success of any software development project hinges on good teamwork. Team Risk Management (see Figure 19) brings together groups with diverse and sometimes conflicting objectives and allows them to discuss risks that cross project boundaries, figure out which risks are most important to mitigate, and pool resources to lower the risk exposure facing all parties.
- **continuous process:** The risks to an organization change constantly. The key to instituting a continuous process is to maintain the surveillance of risks as they change and to constantly identify new risks. Warren Keuffel warns against complacency:

*A software development project's environment is constantly changing, as a result of competitive pressures, organizational strategy and personnel changes, and technical challenges. Too frequently, a risk management plan, like a system architecture design, is prepared at the beginning of the project and then shelved. To be valuable, risk management needs to be revisited as frequently as schedule and technical issues [Keuffel 1999a].*

### Defining Principles

Three defining principles help to ensure success when working in a complex, multi-project environment:

- **forward-looking view:** Everyone is focused on the same tomorrow. Inevitably, when dealing with multiple, interrelated projects that focus on the same success point, conflicts will arise. The organizational management will need to keep all parties focused on the same success point.
- **global perspective:** Everyone has the same definition of success. From an organizational point of view, it may mean lower operations and maintenance costs over a 20-year period. It's organizational management's responsibility to articulate the business goals clearly and define success from a global perspective.
- **shared product vision:** Everyone sees the final capability as the same thing. The entire group of related developers, technical managers, and organizational managers must have a clear vision of where the organization is headed.

### Aspects Peculiar to Product Lines

Product line efforts require a great deal of coordination across project boundaries, which makes well-defined, organizational risk management practices essential. As the organization orchestrates the product line effort, it identifies risks that affect multiple core asset developers and/or product developers.

The core asset and product development teams will manage the risks to their individual projects. (For more information, see the "Technical Risk Management" practice area.) Each project management team will have cost, schedule, and technical objectives that are peculiar to the task they are trying to accomplish. When risks cross multiple projects or affect the organization's successful implementation of a product line approach, the risks should be managed at the organization level.

The multiple viewpoints endemic to product lines will surface in risk management and must be reconciled. For example, core asset developers may define success as meeting delivery dates, whereas product developers may define it as integrating the product and delivering capability to the end user. Both viewpoints are valid; the risk management process needs to "hear" both of them.

Sometimes two stakeholders on the same side of the product fence will have different viewpoints. For example, one military organization decided to use a product line approach to provide situational awareness tools to both embedded weapons systems and command and control systems. One embedded weapons system developer was concerned about weapons system safety and performance in a battle environment. This system developer could not accept immature products or those that weren't rigorously tested by the core asset developers. In the course of managing the project, the developer identified the following risk:

*The core asset development team does not test the situational awareness core assets adequately; we may receive immature products with major defects that can impact the operational effectiveness of our weapons system.*

A command and control development team wanted increased functionality and was willing to accept early deliveries of "beta" version core assets. The team promised its customers incremental deliveries of new capabilities to support an upcoming operational test and identified the following risk:

*The development process used by the core asset development team takes too long to support our time-to-field requirements; we may miss critical delivery commitments.*

These two risks, identified by separate product development teams with different project objectives, point to conflicting mitigation approaches. Using a team risk approach, each of these individual risks would be elevated to the product line organization level, and mitigation strategies would be developed to address both concerns. Those tasks are accomplished using a structure wherein personnel from multiple projects work together to share information about risks that may affect the other project(s) or the product line itself [Gallagher 1999a]. The risk at the product line organization level might be

*The development process used by the core asset development team is rigid, forcing a "one-size-fits-all" delivery and integration approach; the delivery of core assets to reusers may not support project-level objectives and ultimately fail to meet the user's needs.*

The complexity of managing the delivery of core assets to product development teams requires a management structure with mature methods and tools that can arbitrate conflicting needs and develop "win-win" solutions for the entire organization. Organizational risk management practices must provide proactive management methods and tools to help solve the complexities associated with implementing a product line approach.

## Application to Core Asset Development

Core asset development teams should actively and continuously identify and manage risks associated with developing core assets for the product line. (For more information, see the "Technical Risk Management" practice area.) When the risks also affect product development teams or other core asset development teams, they should be elevated, using a team risk approach, to the organization level. Core asset development teams need to understand the objectives of the organization's product line approach and work closely with product developers, the developers of other core assets, and product line managers.

## Application to Product Development

Product development teams should actively and continuously identify and manage risks on their projects. (For more information, see the "Technical Risk Management" practice area.) When risks cross project boundaries and affect other product development teams or core asset development teams, they should be elevated, using a team risk approach, to the organization level. Product development teams need to understand the objectives of the organization's product line approach and work closely with core asset development teams, other product developers, and product line managers.

## Example Practices

The example practices outlined in the “Technical Risk Management” practice area apply here as well. The difference is that the scope of organizational risk management activities transcends individual development projects.

**Team Risk Management (TRM):** TRM, shown in Figure 19, is one cross-organizational approach [Dorofee 1994a]. It was developed originally to allow individual and shared risk management between a government organization and a contractor organization. However, the basic principles are applicable to two projects or organizational structures that have mutual interests and shared risks. Simply stated, two or more projects could install risk management practices following the principles described in this practice area or the example practices described in the “Technical Risk Management” practice area. TRM describes how to establish an overarching risk management process to share risk management activities where appropriate and insure that project-level risks are surfaced at appropriately high levels of management.

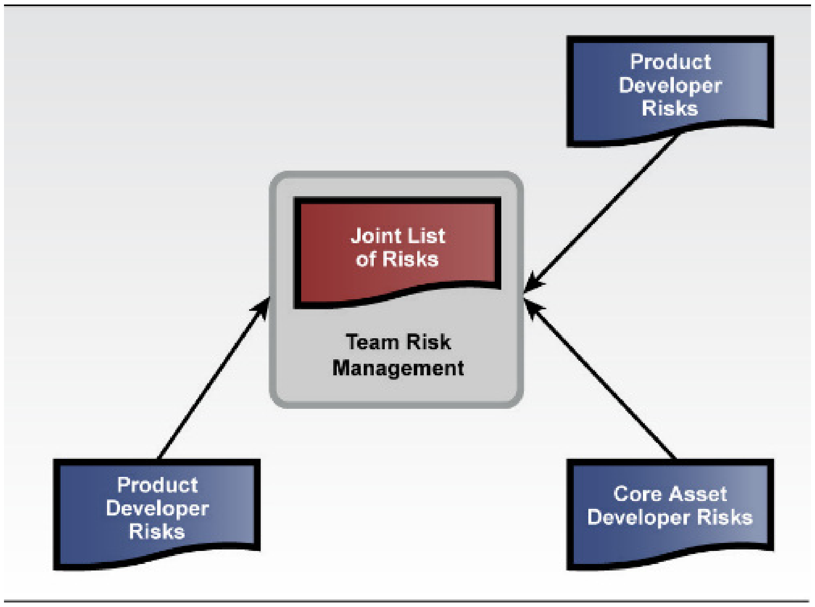


*Figure 19: Team Risk Management*

**Practice area risks:** Risk management activities often gain a head start by asking probing questions designed to test for the presence of risks that often occur in similar situations. In a product line organization, there are known risks associated with each practice area. Those risks can be used as the basis for probing questions associated with risk identification.

**SEI Product Line Technical Probe (PLTP):** The PLTP, described as an example practice under the “Launching and Institutionalizing” practice area, is a way surfacing a product line organization’s strengths and challenges at an organizational level. When applied during a product line launch, the organizational risk management team should track and manage the uncovered challenges as risks.

### Practice Risks

**Non-conducive environment:** The biggest risk in managing uncertainties is creating an environment in which team members feel that they can't talk about risks. In his book on software disasters, Glass points out that one shared characteristic of failed projects is the inability of project members to communicate potential problems to the decision makers within a project. Seventy-two percent of failed projects had team members who knew of impending doom, while only 19% of the project managers on the same projects shared their insights [Glass 1998a]. Risk management allows team members to discuss potential problems in a structured, non-threatening manner providing insight to decision makers.

The risks highlighted in the "Technical Risk Management" practice area also apply.

### Further Reading

[Dorofee 1994a]
Dorofee and colleagues provide an essential resource for starting and running a risk management effort.

[Glass 1998a]
Glass gives compelling case studies of failed software engineering projects that show what risk management could have averted. Glass imparts lessons learned and abstracts common themes and root causes.

## Structuring the Organization

This practice area is about how the organization forms groups to carry out the various responsibilities inherent in a software development effort. All organizations have a structure, even if it's only an implicit one, which defines the roles and responsibilities appropriate to the organization's primary activities. Particular organizational structures are chosen to accommodate enterprise goals and directives, the culture, the nature of business, and available talent. The organizational structure reflects the division of roles, responsibilities, and authority.

In a traditional organization (one that isn't oriented to product lines), individual product teams tend to be fully responsible for technical decisions that affect their products. Even in a matrixed organization, once engineers join a project from their specialty groups, the projects run independently. Specialization might require each product to have its own development group, with no sharing of personnel. The role of the organization's management in this case is to gather and allocate resources and provide high-level oversight—all with the end goal of supporting product teams.

### Aspects Peculiar to Product Lines

A product line approach entails new roles and responsibilities related to the creation of core assets and of products from those assets. This practice area deals with placing those roles into the appropriate organizational units to most effectively support the product line approach.

In a product line context, the dual development of core assets and products dictates an organizational structure that is not product-centric. Beyond that, management must be concerned with identifying the organizational charter and boundaries, identifying functional groupings, allocating and assigning resources, monitoring organizational effectiveness, improving organizational operations, establishing interorganizational relationships, and managing organizational transitions.

Product line organizational structure goes hand in hand with product line operations (see the "Operations" practice area), because it is the embodiment of the roles, actors, and responsibilities defined there. Structuring the organization is like assembling a symphony orchestra—determining how many of each type of instrument will be in each section, selecting the musicians, and assigning people to the appropriate chair in their sections. "Operations" is the practice area that provides the music everyone will play together and guides them through performances. Both practice areas, obviously, are necessary. They result from an overall vision of how the product line will operate on a day-to-day basis. The job of organizational structure is to make sure that each responsibility and function of the product line operations has a work unit in which to reside.

An organizational structure should be chosen to assign at least the following tasks to the appropriate organizational unit or units:

- Determine the production strategy (see "Core Asset Development").
- Determine the product line scope and associated business case and refresh both in a routine and ongoing way.
- Produce and maintain the architecture for the product line.
- Determine the requirements for the product line and product line members.
- Design, produce, and maintain the product line's core assets and associated production plan.
- Assess the core assets for their utility and guide their evolution.
- Produce products.
- Determine the processes to be followed and measure or ensure compliance with them.
- Maintain the production environment in which products are produced.
- Forecast new trends, technologies, and other developments that might affect the future of the product line.

## Application to Core Asset Development

A first-order choice that must be made when choosing an organizational structure is where to assign the people who develop, maintain, and evolve the product line's core assets. Typically, organizations take one of two approaches: (1) they form a separate unit for the developers and maintainers of core assets or (2) they house that effort in the same unit or units that build products [Bass 1997a, Bass 1998a].

Both approaches have their advocates. On the one hand, having a dedicated core asset development group makes the core asset developers one level removed from the pressure of project deadlines, making it more likely for them to produce assets that are generic and not too biased toward the individual product that happens to be under development at the time. Their loyalty is to the product line, not to any one product. On the other hand, having product engineers produce the core assets ensures that the core assets will be truly useful for products and ensures that people who build the core assets are not too far removed from the day-to-day realities of product production. In either case, there must be someone with a cross-product perspective and authority who can identify useful assets for the core asset base and encourage (if not direct) the appropriate product group to produce each core asset for use by other products.

When choosing whether to establish a separate core asset group, consider these factors:

- **the size of the effort and the number of products:** In a product line with many product groups and/or a large number of developers, distributing the core asset task results in an untenably high number of communication channels: every product group will have to talk to every other one. In this circumstance, it helps to have a dedicated core asset group.
- **new development or mostly legacy-based development:** In product line efforts where the core assets are built based on mining legacy systems, it makes more sense to have product developers (who will probably be more familiar with the legacy systems) be responsible for mining core assets that will be generic and fit the scope of the product line.
- **the funding model:** Funding a core asset development group can be problematic. Who pays for it? When working from legacy systems or when the product line approach matures, it may be hard to justify a separate asset development group when the product development groups are adding product-specific features to the core assets.
- **the high or low effort of tailoring core assets:** How much development has to be done to get from the core assets to the products? If the amount of tailoring and new development is small, it may make sense to have most people work in a dedicated fashion on the core assets. If producing products requires substantial tailoring and new development, the asset development job is small by comparison, and integrated groups may be the answer.
- **the volatility of core assets:** Having core assets that evolve frequently and substantially argues for having a dedicated group to manage them, rather than overwhelming the product builders.
- **parallel or sequential product development:** If products are built sequentially, it makes sense to have an integrated team working on them. When several product development projects are performed in parallel, there is a stronger need for a separate core asset group to avoid the multiple redevelopment of the same functionality.

An evolutionary organizational structure resolves the choice essentially by having it both ways. The approach starts the product line effort without a core asset group so that the engineers can assume the full responsibility for turning out a real product under the product line paradigm. However, as soon as there are parallel developments in progress, product development is separated from core asset development. Other approaches that seek a middle ground include frequent staff rotation and processes that require close communication among the groups.

Whichever way this issue is decided, the product line core assets must be managed to bring long-term benefit to the entire organization, rather than to just one specific application. The organizational structure chosen for product line production must assign the decision-making responsibilities for these functions.

And while managing the architecture and other reusable software-related core assets is an obvious product line responsibility that must be assigned to the right organizational unit(s), the management of non-software core assets must be allocated also to achieve the full benefit of core asset reuse. For example, where, how, and by whom will core assets such as product plans, schedules, and budgets be maintained for use across the product family?

### Application to Product Development

Product line systems are developed and managed according to a life cycle that differs from the norm. Building products from core assets places greater emphasis on software integration and the testing of interfaces across components. For the unit or units assigned to product development, responsibilities include the following:

- making sure that each new product uses the core asset base according to the production plan
- working with the core asset owners to evolve new capabilities if the core assets are deficient in some way for a new product
- providing feedback to the core asset developers concerning the suitability and quality of the core assets

The product unit(s) may also negotiate customer requirements to situate new products within the scope of the product line to the greatest degree possible, although some organizational structures assign the management of product line requirements to their own units.

The organizational structure must assign responsibility for these roles, as well as for the more traditional product development roles.

### Example Practices

**Organizational models from a Swedish product line survey:** Jan Bosch described four separate organizational models [Bosch 2000b] after studying a number of product line corporations. He discriminated between the models based only on organizational size and did not take into account other factors. The four models he identified were

- **development department:** In this model, all software development is concentrated in a single unit. Each member of the unit is expected to be a jack-of-all-trades in the product line, performing core asset development tasks or product development tasks when appropriate. This model appears in small organizations and those that provide consulting services. Although it is a simple model with short communication paths, it has a number of distinct drawbacks. Bosch wrote that it probably works only for units of up to 30 people. But in very small organizations whose product lines are commensurately small, it can be a viable starting-out approach.
- **business units:** Each business unit is responsible for a subset of the systems in the product line, which are clustered by similarity. Core assets are developed by the business units that need them and made available to the community; collaboration across business units to develop new core assets together is possible. This model has variations depending on how much flexibility a business unit has in developing (or modifying) a core asset. With no constraints, the products tend to spiral off on their own evolution paths, negating the product line approach. At higher levels of

maturity, the responsibilities for particular core assets are assigned to specific business units. These units are constrained to maintain their core assets for the general use of the entire product line, and other business units are required to use them. Bosch estimates that this model could apply to organizations with 30 to 100 employees. However, this model suffers from the obvious risk, mentioned above, that a business unit will focus on its own product(s) first, and the good of the product line will take a back seat.

- **domain engineering unit:** In this model, a special unit is given the responsibility of developing and maintaining the core asset base. Business units build the products using those core assets. Bosch writes that when organizations exceed 100 employees, the *n*-to-*n* communication channels among separate business units become untenable, and a focusing channel to a central core asset unit becomes necessary. In this model, a strong and disciplined process becomes much more important in order to manage the communication and ensure that the overall health of the product line is the endgame of all parties.
- **hierarchical domain engineering units:** In a product line that is very large and/or very complex, it may pay to regard it hierarchically. That is, the product line may consist of subgroups that have more in common with each other than with other members of the product line. In this case, one core asset development unit may turn out core assets for the product line at large, and another may turn out core assets for the specialized subgroup. This example has only two levels, but the model could be extended indefinitely if the subgroup has a specialized subgroup within it and that specialized subgroup has another within it and so forth. This model works for very large product lines built by very large organizations. Its main disadvantage is its tendency to bloat, reducing the organization's responsiveness to new needs.

**Organizational structure for a reuse business:** Jacobson, Griss, and Jonsson call for organizing a reuse-based business around a set of competence units, which contain "workers with similar competencies and entity object types that these workers are responsible for" [Jacobson 1997a, Ch. 9]. They prescribe the following units:

- requirements capture unit
- design unit
- testing unit
- component engineering unit
- architecture unit
- component support unit

The component support unit is responsible for "packaging and facilitating the reuse of component systems, and [is] mostly concerned with maintaining the facades and distributing the component systems so that reusers can access the desired components" [Jacobson 1997a].

Workers from these competence units are drawn to form an application family engineering team, one or more component system engineering teams, and a group of application system engineering teams. Application family engineering is where product line decisions are made such as which applications to develop and when. It also crafts the broad product line architecture. Component system engineering is responsible for the detailed architecture and design of a component system. Application system engineering builds products for individual customers.

**Small, distributed core group with component representatives:** In choosing whether the core asset group is separate or distributed throughout product groups, at least one company we know has opted for a hybrid approach. This company, which has about 50 employees building a product line of information systems, has a set of product groups that revolve around a small but central core group. In this model, the core group is responsible primarily for common support activities such as configuration management, maintenance of the core asset base, some process definition, and the like. An architect for the entire product line would also normally reside in this group. However, in this company, the architecture is not being evolved, and the architect has been reassigned. In each product group, a component representative has the joint responsibility of ensuring the use of core components and creating and evolving the software core assets. A strict protocol is enforced in which any candidate for inclusion in the core asset base must have at least two component representatives sponsoring it. This protocol ensures the asset's reusability across at least two products.

**Structure for a distributed or globally dispersed organization:** Work force distribution and globalization present additional challenges for organizing a product line effort. Some organizations are addressing these challenges in these ways:

- virtual core asset development teams with leaders in one location and dedicated core asset developers in other locations
- core asset development teams in one location and product development teams in other locations

## Practice Risks

- **choosing a structure that is inappropriate for the organizational context at hand:** An inappropriate structure—one that doesn't match the talent base and/or culture of the organization—will cause the organization to fail to achieve the compromise between building overly generic core assets that are too general or expensive to serve any specific product and product-specific software that is too customized to serve in a product line.
- **lack of ample feedback and communication mechanisms:** Any organization's structure requires ongoing monitoring. These mechanisms are necessary to insure that the chosen structure is working as intended or to raise signal flags when it isn't.
- **one size fits all, for all time:** Many organizations report changing their organizational structure as their product line organizations mature. If your organizational structure is no longer adequate, it doesn't mean that it was wrong—it may just be a sign that it's time to change it.
- **reorganizing too often:** On the other hand, nothing seems to disrupt an organization like a reorganization. Make sure it is necessary before putting everyone through that stress.

- **ignoring key success factors:** Some key success factors in implementing structural change according to Myers, Maher, and Deimel[14] are described below. Failure to account for any of these factors constitutes a risk.
  - **the current level of organizational stress:** How much stress is the organization currently under as a result of previous changes or other stress-inducing factors?
  - **the implementation history:** How effective or ineffective has the organization been in effecting change in the past?
  - **sponsorship:** How effective is the key executive in obtaining commitments to change, communicating the support for change, and managing change?
  - **resistance management:** How will the organization address the inevitable resistance to change?
  - **culture:** Does the proposed change conflict or align with the organization's values, behaviors, and unwritten rules?
  - **change agent skills:** Are the change managers and staff equipped with the skills and motivation needed to implement the change?
- **adopting an organizational structure without an accompanying operational concept:** An organizational structure is just a wiring diagram. Without a clear articulation of the roles, responsibilities, and day-to-day operating procedures, any organizational structure will fail to meet its product line needs.

### Further Reading

[Bosch 2000b]
Bosch describes four product line organizational models.

[Brownsword 1996a]
Brownsword and Clements describe stages in the evolution of CelsiusTech's organizational structure as its software product line matured and organizational needs changed.

[Jacobson 1997a]
Jacobson, Griss, and Jonsson describe an organizational structure based on the concept of competence units.

## Technology Forecasting

Ironically, one of the most insidious dangers facing an organization is long-term success, for success breeds complacency. While routinely cranking out products and congratulating itself on its stellar productivity figures, an organization is vulnerable to being blindsided by a competitor who is sporting new features, new ideas, and new technologies. To head off such calamities, an organization must institutionalize vigilance, and one way to do that is by practicing technology forecasting. Technology

14 Myers, C. R.; Maher, J. H.; & Deimel, B. *Managing Technological Change*, Version 1.92. SEI Workshop given in 1990.

forecasting helps provide strategic market planning—that is, identifying trends and predicting what the relevant markets will bear [Ryans 2000a]. It spots emerging standards, allowing an organization to position itself early to lead, or at least react, with agility. It reduces risk with respect to innovations, provides the basis for planning and directing investments in research and development areas, and helps set the direction of product migration.

Technology forecasting helps take the pulse of the core technologies on which the products rely, as well as the tools, techniques, methods, and processes used to develop the products and bring them to market. Development techniques and tools are particularly important when time to market matters.

Technology is forecast in two areas:

1. technologies that support internal software development, which includes tools, processes, and methods for producing the software that will end up in products. These technologies may include
   - development paradigms and notations, such as the Unified Modeling Language (UML)
   - Web-based and wireless capabilities and languages such as the Extensible Markup Language (XML)
   - code development suites and environments
   - analysis tools
   - planning, configuration management, and requirements management techniques and tools
   - process improvement and process management approaches
2. customer solutions, meaning technologies that will affect (or end up as) features or capabilities embedded in products. These technologies may include
   - more efficient hardware platforms
   - better software platforms (such as databases or network and communications middleware)
   - improved user interfaces
   - faster database-searching strategies
   - new user-oriented paradigms (such as visual environments and Web-based interfaces)
   - improved architectural solutions
   - better problem-reporting systems
   - emerging standards
   - new features that enhance the capabilities of users

The rapid pace of technology change makes assessing new technology very challenging [Brown 1996a] and limits the technology time horizon to no more than three to five years. Consequently, plan to update your technology forecasts periodically. Christensen describes disruptive innovations—technologies that are not anticipated and have pervasive impact on entire markets [Christensen 1997a]. While disruptive innovations will likely elude technology-forecasting activities, most other technology changes can be anticipated by proactive looks across the domain and the software technology horizon.

## Aspects Peculiar to Product Lines

For non-product-line development, technology forecasting may be performed infrequently. More typically, technology "now-casting" is done, meaning that the best currently available technologies are identified and assessed for their immediate applicability to the situation at hand and likely near-term changes. In product lines, however, technologies are identified and assessed continuously. They are assessed both for their immediate benefit and their potential future applicability. A technology that shows promise today may find its way into the product line in three years, or it may flicker out before then. On the other hand, a technology deemed much too risky and on the edge today might be the foundation for the next revolution. In product line technology forecasting, time is an ally, and we can afford to be more patient than we could be in single-system development. And since we can amortize the cost over more products, we should also be able to be more thorough.

Technology forecasting for product lines covers both technologies that enable specific product features and technologies that support the engineering tasks in the development of those features. The former category will depend on the application domain and is too general for specific advice. The latter category includes

- software technology that supports variation modeling, automation, or producability
- process innovations that make the management, production, and deployment of new products or product releases more efficient and predictable
- configuration management strategies and techniques that help the product line organization provide better customer services through the ability to recreate and service the "instance" of the product installed and the upgrades that are appropriate for that installation
- trends within the relevant standards bodies and the migration of the technical basis of the standards that apply to the product line. Ideally, the product line organization exploits the technology forecast to achieve a technology leadership position.

The product line's technology forecasts are an important input into the development of the production strategy and the production method. These forecasts guide the development of forward looking strategies that define a migration path for integrating emerging new product construction technologies into the production method of the product line. The production strategy should be continually updated as technologies mature or fade and the long-term approaches to core asset and product development change. Revisions of the technology forecasts should signal the need to revisit the production strategy.

## Application to Core Asset Development

Earlier in this practice area, we distinguished between technologies that benefit internal development and those that are customer focused. While both are relevant for core assets, the technologies that impact development require special emphasis as core assets. Tools and techniques, such as more efficient configuration management or improved tool integration, can be leveraged over a variety of products to improve efficiency and quality; these tools and techniques offer a strong strategic advantage. The architect and developers are primary stakeholders for these types of innovations.

Technology forecasts also directly aid the core asset designer in selecting variation points and variability mechanisms. The forecasts describe how radically technology is likely to change as well as which technologies will be most subject to change. The designer can determine how much flexibility is required at each variation point and select the variability mechanism accordingly. For example, minor changes that occur infrequently can be handled by a static mechanism such as class inheritance. Changes that occur more frequently may require a dynamic mechanism that can be automated or applied by the product user.

### Application to Product Development

Technology forecasting increases the likelihood that products will be useful to customers by predicting those features that will become more (or less) desirable. The forecasts also help identify those features most likely to be affected by changes to their underlying implementation technologies. Both types of forecasts help identify variants that can be anticipated at variation points and ultimately when adjusting the product line scope.

### Example Practices

**Technology forecasting as continuous improvement:** A continuous process improvement paradigm underlies many of the practices of technology forecasting. This paradigm suggests the need to constantly monitor the current technologies, tools, and processes that make up business practices in order to uncover opportunities for improved practices. The challenge is to focus the technology forecast on the areas offering the highest potential return to both the product line organization and the customer. Opportunities for improvement may result from the continuous analysis of defects and trouble reports yielding insights into the areas where the product line could leverage technology improvements. Additional opportunities for improvement may surface as a result of an analysis of changes in the marketplace that have implications on new technology.

**Technical steering group:** A technical steering group is a proven mechanism for keeping current with technology trends. This type of group comprises senior technical managers who analyze new trends, customer needs, and technologies. One organization we know also appointed a group of senior engineers as “technology stewards” and tasked them with accumulating and maintaining sufficient expertise in various technical areas so that they could become reliable forecasters.

**Technology sources:** Once the product line organization has a rationale and focus for conducting the technology forecast, the next task is to identify the sources of technology that are relevant to their needs. Sources include

- centers of technology investigation, such as the Software Engineering Institute (SEI), university laboratories, and corporate research and development establishments that publish their results
- research and development conferences, such as the Software Product Line Conference (SPLC), the International Conference on Software Engineering (ICSE), the conference on Object-Oriented Systems, Programming Languages, and Applications (OOSPLA), and others
- networks of professional associations and associates
- journals and periodicals

**Validating the forecast:** Once focused on the forecast objectives, specific steps include the following:

- Perform sufficient research to isolate the subset of promising technologies.
- Validate the technologies through simulations or models to focus on the most promising technology.
- Analyze and quantify the benefits for both developers and customers. Determine any adverse impacts.
- Conduct pilot tests as a proof of concept.
- Integrate the technology with other assets and then conduct testing.
- Provide adoption training and rollout.

### Practice Risks

Inadequately forecasting technology results, predictably, in new and promising technologies passing you by. That, in turn, results in a product line that is obsolete before its time and opens the door for a more attentive competitor to steal your market. An inadequate technology forecast can result from

- **forecasting that is driven by technology rather than by business needs:** Analysts may become enamored by the technology and the search for it and lose sight of the corresponding business requirements.
- **ivory-tower forecasting:** Technology forecasting can become a "sandbox" or an "ivory tower" that is so removed from the day-to-day operations that crucial issues, objectives, and priorities get lost.
- **a lack of purpose:** If the problem that the product line organization is trying to solve is unknown, the search for new or emerging technology to maintain product line leadership has the potential to be limitless.

An inadequate forecast can result in

- **the wrong technology choice:** The wrong product gets built, and the market turns elsewhere for solutions. That can be caused by making investments in the wrong technology or an obsolete one or by simply choosing "what's good" over "what's popular."
- **an architecture that can't support new technology:** Radical changes in technology that the architecture can't accommodate will result in the obsolescence of the product line. The product line organization has a lot riding on its architectural decisions and needs as much advance warning as possible about any radical technological changes that could undermine them.

### Further Reading

[Brown 1996a]
Brown and Wallnau describe how to position candidate technology in the context of your organization's needs and capabilities.

[Christensen 1997a]
Christensen examines changes wrought by new technologies, most specifically those that are disruptive innovations, which dramatically change entire markets.

[Ryans 2000a]
Ryans and colleagues describe a strategic marketing planning approach for technology organizations. The emphasis on adapting to new market conditions and changes in the competitive climate are relevant to this practice area. Examples from Intel, Compaq, Hewlett-Packard, Glaxo-Welcome, and General Electric provide insights from industry leaders.

## Training

Training is a core activity of any software development organization. The purpose of it is to provide the skills and knowledge needed to perform software management and technical roles. A training program involves identifying the training needed by the organization and the entities within it and then developing or procuring that training. Thus, as in many other practice areas, there is an initiation or planning phase followed by an execution phase. Training can be informal through mentoring or other on-the-job mechanisms or formal through classroom instruction or video sessions. Adequate resources and funding are needed if the training is to be effective and should be documented in a training plan.

### Aspects Peculiar to Product Lines

Training is an element of both the initial product line adoption and the longer term product line evolution. This practice area focuses on the training practices that need to be instituted by management to ensure that the organizational units responsible for creating, fielding, and evolving the product line have properly trained personnel.

Management’s support of training includes

- committing to an appropriate training plan
- ensuring that the plan is implemented and that the training is monitored for effectiveness
- ensuring that the product line training is consistent with and supportive of the overall product line adoption process or any process-improvement efforts

An organization’s approach to training in product lines must focus on establishing a core competence in the creation and usage of core assets. Thus, it is not enough, for example, to send people to a course in object-oriented design or software reuse and then expect them to build product lines. All training must occur within the context of the organization’s adoption plan for product line practices and address the skills needed by people for the new roles they will assume within the organization as it moves away from the single-system, project-centered view to the multi-system, product line view.

Product line training must be viewed as a strategic activity that should be planned accordingly. A training plan should align with the overall product line adoption plan and tie training goals to the business goals of the organization. For example, if a business goal is to reduce the time to market of products in the product line, any training in software reuse practices must emphasize the creation of specifically targeted reusable assets rather than an opportunistic reuse library. In the product line context, reuse is a means to an end, not an end in itself, and reuse training must focus on designing for commonality and controlling variability rather than on creating class libraries.

The appropriate product line training also depends on the current state and experience base of the organization. For example, an organization already fairly sophisticated about architecture and architecture-based design will have less of a need for training in those areas.

### Application to Core Asset Development

Training for core asset development is primarily training in the software engineering practice areas. Any such training should be preceded by an introductory course that explains product line concepts in general and the organization’s planned product lines in particular. The specific training associated with each software engineering practice (for example, training in a specific domain-analysis method or training in architecture definition) must be tied to the goal of creating a core asset base to support a product line. Similarly, training on the tools for representing and documenting the outputs of a domain analysis or a software architecture definition effort should focus on complementing the analysis and design skills of the core asset creators rather than their coding skills.

If externally available software (for example, COTS components, Web services, or open source software) is to be acquired as part of a core asset strategy, the training should focus primarily on how to choose and integrate software that makes sense for the product line. Similarly, if legacy software is to be repackaged as a core asset for the product line, the training should focus primarily on how to analyze its reusability rather than on how to “wrap” it for inclusion in a current product.

Finally, training materials and plans make first-class core assets themselves.

### Application to Product Development

Product development training is the complement of core asset development training; its primary goal is to ensure that product developers know how to create products in the product line from the core asset base. It, too, should be preceded by an introductory course in software product lines with a particular focus on the organization’s product line. The emphasis is on the effective use and reuse of core assets such as a domain model and architecture following the dictates of a production plan. The architecture for a single system, for example, is derived from (if not identical to) the product line architecture and not built from scratch; the training must emphasize the need to follow the production plan that was established for the product line. One of the major “themes” of the product development training should be that core asset usage is strategic in nature and that core assets that appear to be less than optimal for a *particular* product may be an optimal element of the *overall* product line strategy.

Another important aspect of product development training is getting people to follow the process defined for using core assets and correcting problems. For example, problems with core assets should always be referred to the core asset creators rather than the product developers. That way local “fixes” proposed for a particular customer can be assessed against the long-term needs of the product line (for example, the configuration management of core assets or control of variations).

## Example Practices

**Develop a training plan:** The most important elements of product line training are the identification of training needs and the creation of a strategic training plan to meet those needs. This plan must identify the current skill gaps and determine the training requirements needed to fill them. It must also address how those skills will be established and maintained in the teams that build product lines: in-house courses, external courses, on-the-job training, mentoring, and so forth. Creating and implementing such a training plan is a key element of the cultural change needed for product line adoption.

**Train people for the transition to a product line approach:** Training to prepare people for the transition to product lines includes such elements as

- an introductory course on product line concepts and terminology
- an overview of the organization's current and planned product lines
- an overview of the proposed development process, including changes in existing processes, organizational structure, and roles
- a presentation of the concept of operations (CONOPS) for the product line to explain the goal state of the organization and the role of training in achieving that state
- training in specific product line practices or concepts. Here, software architecture deserves a special mention as a concept whose role and use should be emphasized, especially among the product builders. A familiarization course on the particular architecture being used as the foundation of the product line (including the ways it supports variability) is often most useful.
- training in the production techniques that automate the production of products from core assets, as described in the production plan
- training in supporting technologies (such as tools for representing and documenting the core assets)

Note that the first four items above are really about education rather than training and that even experienced practitioners may need to be reeducated about how their skills and roles apply in the product line context.

Examples of organizations creating educational and training materials specifically for software product lines include

- the Software Engineering Institute (SEI), which offers a software product line curriculum with courses covering introductory topics, product line adoption, product line development, and diagnosing an organization's readiness to adopt or ability to succeed with a product line approach (www.sei.cmu.edu/productlines/spl_curriculum.html)
- the Fraunhofer Institute for Experimental Software Engineering (IESE), which offers a set of services, based on the IESE Product Line Software Engineering (PuLSE) method, for setting up and running a product line organization (www.iese.fraunhofer.de/PuLSE/)

Training needs to be tailored to the specific processes and skills of the organization and to the product line adoption plan. Establishing a core competence in core asset creation and usage means understanding and applying new concepts, not getting high marks in a training class.[15] Any training in, for example, domain analysis, software architecture, object-oriented technology, model-driven development, design patterns and frameworks, specific programming languages, or specific development environments must be planned and implemented as part of the product line strategy and not as ends in themselves.

**Implement the training plan:** Implementing the training plan includes making decisions about the most effective way to deliver the training: classroom training, hands-on training, tutorials, workshops, pilot projects, mentoring, and so forth. An important decision in this regard is the scope of any planned in-house training and mentoring capability and the extent to which external courses and instructors will be used. In general, implementing the product line training plan involves some or all of the following choices:

- augmenting current training activities to support product lines
- replacing existing training activities
- adding new training activities

To ensure that the training meets the goals established for it in the training plan, it must be monitored and measured. Any lessons learned about the timeliness, relevance, level of difficulty, deficiencies, and effectiveness of the training should be collected from the trainers and trainees and incorporated into future training.

## Practice Risks

An inadequate training program results in a staff that is ill-prepared to perform their jobs efficiently and effectively in the product line organization. Component developers, for example, who are not trained to create truly reusable assets will probably not do so and will consequently undermine the quality of the core assets.

Inadequate training can result from

- **a sink-or-swim approach:** In this approach, the organization assigns ill-prepared people to perform product line tasks and has to scramble later to fill the educational gap.
- **inappropriate focus:** Too much time spent training people on the tools, programming language, or development environment without the requisite product technology foundation will only succeed in helping to automate the wrong operation.
- **a lack of strategic investment:** Sacrificing vital training in order to meet current customer schedules and deliverables will, in the long run, result in staff who are ill-equipped to handle the needs of the product line operation.

[15] CelsiusTech found traditional classroom performance to be an unreliable predictor of post-training performance on real projects [Brownsword 1996a, p. 62].

- **a lack of the big picture:** If insufficient time is spent on communicating the product line vision and on building an awareness and acceptance of the product line, the rest of the product line training will fail to have a meaningful context.
- **inadequate training resources:** A lack of the necessary instructors, funding, classroom facilities, hardware, or software can derail any training plan.
- **a lack of coordination:** If the training plan is not coordinated with the overall product line adoption plan or process-improvement plan, the staff will probably become frustrated and overwhelmed.
- **a lack of training assessment:** If the effectiveness of the training is not monitored or measured, there will be no basis on which to predict the results or improve the training.

### Further Reading

[Brownsword 1996a]
Brownsword and Clements describe how CelsiusTech approached the issue of skills and knowledge, beginning with the formative years of its ShipSystem 2000 software product line that was launched in early 1986.

[Goldberg 1995a]
Goldberg and Rubin devote two full chapters to training in object-oriented technology. Chapter 18 describes what a training plan is and discusses it in terms of both training and educating the entire team. The discussion covers subject areas, proficiency levels, and training formats (for example, classroom, mentoring, and self-study). Chapter 19 describes how to set up a training plan and provides several examples of real training projects.

[Lim 1998a]
Lim devotes Chapter 16 to staffing software reuse projects. It includes an identification of the key roles and responsibilities for implementing reuse and a description of the elements of a core curriculum in reuse.

## Frequently Asked Questions

1. **Concepts and Terminology**
   - Isn't a product line just the group of products produced by a single business unit or profit/loss center?
   - If a product line is a set of products, does that mean I have to build them all?
   - What's the difference between a domain and a product line? For instance, I know there is a telecommunications domain, and I've heard of a telecommunications product line. What's the difference?
   - Isn't software product line practice the same as single-system development with reuse?

- Vendors and other developers issue subsequent releases of single products all the time, and have been doing so for years. Each release is built in the same way-you would say from the same core asset base. Technically speaking, what's the difference between a software product line and multiple releases of the same product?
- Isn't product line practice just another name for domain engineering?
- Isn't product line practice just another name for component-based development?

2. **Are Product Lines Right for My Organization?**
   - My organization is very small. Can we build a software product line?
   - My organization is very large. Can we build a software product line?
   - How can I make my organization build software product lines when people are busy in their own stovepipes? I don't have enough influence to change the way the organization does business.
   - The SEI Capability Maturity Model Integration® models (CMMI®) puts important aspects of product line practice at Level 4. Do I have to be at Level 4 before I can hope to field a software product line?
   - What are the various economic motivations for a software product line?
   - How long before the approach pays off?
   - We're doing okay. Why should I change the way I do business and undergo the upheaval that a product line will bring?
   - Can you really be competitive in the marketplace with a software product line? Doesn't it take away your flexibility if you're locked into a product line and, say, an architecture?
3. **Exploring the Issues More Deeply**
   - Do successful software product line organizations have certain traits in common?
   - You have one practice area called 'Requirements Engineering' and another called 'Understanding Relevant Domains.' What's the difference between requirements analysis and domain analysis?
   - What is the relationship between process improvement and product line practice?
   - Must all products in the software product line share the same architecture? If my products have different architectures but make heavy use of other shared assets, isn't that a software product line?
4. **Using Software Product Lines with Other Approaches**
   - Can I use open source software packages in my product line?
   - Does the software product line approach work in a globalization environment?
   - Can a product line approach be compatible with agile development methods?
   - Is a model-driven development (MDD) approach compatible with the software product line strategy?
   - Does a system-of-systems (SoS) context rule out the use of software product lines?

- Is there any connection between service-oriented architectures (SOAs) and software product lines?
- How are SOA and software product line approaches alike?
- How are SOA and software product line approaches different?
- Can SOA and software product lines be used together?
- Can the framework provide guidance for a move to SOA?

5. **Product Lines in the Context of Acquisition**
   - We're a U.S. government contractor, and our government customer wants to build a product line by buying core assets from us and then staging an open competition to build products using them. How can we possibly compete on the product-building side when all these new companies enter the fray and claim to be able to build products cheaper, because they didn't have to pay for the core assets?
   - We work for a government contractor that wants to bid on a competition to build products based on core assets developed by another contractor. How can we possibly compete given the intimate knowledge possessed by the asset development contractor?
   - I'm a government contractor, and the government wants me to supply reusable core components for other organizations to use in building a product line. Why should I open myself up to the legal liability? I'll be liable if they use my components incorrectly!
   - What acquisition strategies should a DoD or government organization consider when acquiring a product line? And what is their potential impact on the acquiring organization?
6. **Getting Started**
   - All right, I want to start a software product line. What do I do first?
   - My company builds a broadly related group of products, each of which is a group of closely related products. Should I plan to build a single product line or a group of product lines?
   - I want to pilot software product lines in my organization. What are the criteria for a good pilot project?
   - Where is the resistance to adopting a product line approach usually found?
   - Where can I go to get more information?
   - Where can I read about other organizations that have successfully adopted the software product line approach?

## Concepts and Terminology

*Isn't a product line just the group of products produced by a single business unit or profit/loss center?*

That is what some people mean when they use the term *product line* (see "A Note on Terminology"), but it's not what we mean. A product line is a set of products carefully chosen to take simultaneous advantage of commonality and market opportunities. These opportunities could span what is normally produced by a single business unit. For example, the software for commercial avionics and the software for military avionics could span separate business units in a company but be developed as a single product line by a software group that reports to both. Or, a single business unit may be responsible for developing and fielding more than one software product line. In general, a business unit is formed for organizational or financial reasons and may be responsible for (part of) one or more product lines.

*If a product line is a set of products, does that mean I have to build them all?*

No. While the term *product line* usually brings to mind a specific set of fielded products, you can think of a product line as defining a virtual set of products, each sharing a set of common, managed features. Only those products that satisfy some specific market need are actually built. The others represent untapped capability that you would be willing to build should the opportunity arise.

*What's the difference between a domain and a product line? For instance, I know there is a telecommunications domain, and I've heard of a telecommunications product line. What's the difference?*

A domain is a specialized body of knowledge, an area of expertise, or a collection of related functionality. A product line is the set of software-intensive systems sharing a common, managed set of features that satisfy particular market or mission needs. The telecommunications domain is a set of telecommunications problems, which in turn consists of other domains (switching systems, protocols, telephony, networks, etc.). A telecommunications product line is a specific set of systems that addresses some of those problems. Some people use the term *domain* to refer to a set of systems that employ knowledge in a particular domain, but really that's an incorrect use of the term. It's hard to imagine any nontrivial software system that doesn't encompass knowledge from several domains.

*Isn't software product line practice the same as single-system development with reuse?*

No. It differs in two fundamental ways. First, building a software product line is all about planning a plurality of products that will exist in the field and be maintained simultaneously, not just one that evolves over time. And second, the reuse that's involved is carefully planned, strategic, and applies across the entire set. In fact, one of the primary distinguishing characteristics of product line practice is preplanned reuse rather than ad hoc or one-time-only reuse. Software product line practice encourages choices and options that are optimized from the beginning for more than a single system.

*Vendors and other developers issue subsequent releases of single products all the time, and have been doing so for years. Each release is built in the same way—you would say from the same core asset base. Technically speaking, what's the difference between a software product line and multiple releases of the same product?*

There are some similarities, but the differences are acute enough to matter. The primary difference is that vendors who release subsequent versions try to retire earlier versions as soon as they can. Also, later versions are usually supersets of earlier versions. The latest version is always the one of interest. In a software product line, all versions are of interest, because they provide different functionality and quality attributes, each serving different market segments or missions.

*Isn't product line practice just another name for domain engineering?*

Domain engineering addresses only half of the problem—core asset development and acquisition. Product line practice addresses both domain engineering and product development using the core assets—or what has been called application engineering.

*Isn't product line practice just another name for component-based development?*

Software product lines certainly rely on a form of component-based development. The typical definition of component-based development involves the selection of components from a library or the marketplace to build products. Though the products in software product lines certainly are composed of components, these components are all specified by the product line architecture. Moreover, the components are assembled in a prescribed way; the prescription comes both from the architecture and the production plan. Software product lines involve the strategic use of components. Component-based development usually is missing such a strategic and systematic approach.

## Are Product Lines Right for My Organization?

*My organization is very small. Can we build a software product line?*

Yes. There is no reason why even a one-person software boutique cannot benefit from product line practices. Small organizations are well served by "lightweight" versions of the practices. And some of the organizational and role changes that come with product lines may, in fact, be easier to carry out in a smaller organization. Case studies show that small companies (even start-ups) have understood that in order to field a number of different systems, their severe resource constraints compelled them to exploit all possible commonality among the systems. They used a product line approach to leverage their small staffs and budgets across a successful blend of products. For an example of such a case study, see [Clements 2002c, Ch. 11].

*My organization is very large. Can we build a software product line?*

Yes, and there are many examples of success. The larger the organization, the more important it is to gain control over the costs and procedures for doing business and to effectively manage products.

*How can I make my organization build software product lines when people are busy in their own stovepipes? I don't have enough influence to change the way the organization does business.*

Nobody said it would be easy. It is necessary to make the business case for product lines (see the "Building a Business Case" practice area) and to find a champion above the level of all the stovepipe organizations. One of the missions of the SEI's Product Line Practice Initiative is to provide the ammunition needed to make the business case to potential champions. If case studies are documented and practices are defined within the context of a well-defined framework, convincing the decision makers should be easier. Also, many adoption strategies are possible. For example, you could find a part of the organization where people are open to cross-unit cooperation and begin building a small product line there. Incrementally grow the scope of the product line, and make sure you collect data to record the economies that you gain. The "Launching and Institutionalizing" practice area is especially relevant here.

*The SEI Capability Maturity Model Integration® models (CMMI®) puts important aspects of product line practice at Level 4. Do I have to be at Level 4 before I can hope to field a software product line?*

No, but certain Level-2 and -3 practices are key. For example, an organization must have well-developed configuration management and product planning skills before it can have much hope of fielding a product line. Moreover, successful product line practice requires process discipline. That's why "Process Discipline" is one of the practice areas. Organizations that are at Level 4 can transition to product line practices with fewer risks, but the risks can be managed at lower maturity levels as long as they are identified and mitigated.

*What are the various economic motivations for a software product line?*

Many organizations that have started software product lines in recent years have done so out of economic necessity. They have found that they simply could not continue to do business as they had in the past and still be competitive. Chief among the economic benefits are reduced time to market, greater market agility, opportunities for mass customization, and lower unit costs. These factors are driven by greater productivity from scarce or costly worker resources. Many organizations are finding that they simply cannot find enough qualified people to expand their business without embracing product line practices or afford to hire them even if they were available. In the realm of acquisition, the government may commission a product line to eliminate wasteful duplication among programs or to enable it to assemble more cost-effective systems by more easily obtaining products from a range of vendors. See the "Building a Business Case" practice area for more information.

*How long before the approach pays off?*

That's a hard question to answer, because it depends on too many organization-specific conditions. If you have decided to build products for which there is no market, no matter how efficiently you build them, you will not make money. However, a slightly different form of the question can be asked: How many products are required before building them as a product line is more cost-effective than building them as separate systems? The answer to that question lies somewhere in the neighborhood of two or three systems. That is, if your product set is expected to be populated by three or more systems, you're almost certainly better off to build them as a software product line than as separate systems. (See "It Takes Two," [Clements 2002c, p. 226].)

*We're doing okay. Why should I change the way I do business and undergo the upheaval that a product line will bring?*

First of all, it's not a foregone conclusion that product line practice brings upheaval. Some adoption strategies prescribe starting small and growing incrementally. New technologies are emerging that are allowing companies to quickly extract and manage commonality among systems they've already fielded; these systems can form the foundation for a product line that can be grown over time. But no organization should begin to build a product line without understanding the costs and benefits. Even if you're doing fine now, circumstances that motivate an organization are often based on long-term vision and not the status quo. You should also ask yourself whether your competition is standing still. The choice to adopt product line practices should be a strategy for achieving specific business goals. That's why it's important to build a business case before you decide to launch a product line.

*Can you really be competitive in the marketplace with a software product line? Doesn't it take away your flexibility if you're locked into a product line and, say, an architecture?*

Competitiveness should be sharply increased, because you’re gaining the ability to respond quickly to opportunities that fall within the product line’s scope—a response time sometimes measured in days instead of years. Also gained is increased ability to respond to user needs. What is lost is unbounded variability. The key is to understand which variations are needed and which variation mechanisms can best support them. Early steps are understanding the relevant domains, building a business case for a set of products, and product line scoping. If domain analysis has captured the requirements of the application domain, the product line has been scoped commensurately, and the architecture and other core assets reflect those needs through preplanned variation mechanisms, new products can be brought to market far faster than before with minimal loss of flexibility as compared to one-of-a-kind systems. However, a product line organization needs to beware of complacency and should always keep an eye on the horizon for new technologies, new user needs, and new opportunities that might compel a shift in the product line’s scope to keep it vigorous into the future. Finally, you’re never “locked” into a product line, as the question implies. If a business case so warrants, you can always build a product outside the product line. For example, if an opportunity to enter a new market comes up, that first product can be used to test the waters and could even become the basis for a new product line.

## Exploring the Issues More Deeply

*Do successful software product line organizations have certain traits in common?*

They do indeed. The best ones are comfortable with process discipline, have a strong and tireless champion for the approach in a position of leadership while the effort is being launched, and have long and broad experience in the application areas covered by the product line. Less tangibly but just as observable is one other aspect, something we’ve nicknamed the “e pluribus unum effect.” *E pluribus unum* is Latin, of course, for “out of many, one.” All the most successful product line organizations we’ve observed regard their primary mission as building and maintaining the product line (singular) or (equivalently) a production *capability*. Organizations still climbing the product line hill tend to describe themselves as turning out products (plural). The difference in mindset is subtle but powerful. Companies with the singular outlook realize that they have built a product capability that transcends any one product or even any several products. To them, turning out products is the easy part.

*You have one practice area called ‘Requirements Engineering’ and another called ‘Understanding Relevant Domains.’ What’s the difference between requirements analysis and domain analysis?*

Remember that a domain is a specialized body of knowledge or area of expertise. Hence, domain analysis explores and captures the knowledge areas key to a product line and therefore tends to be broader in scope than requirements analysis. Requirements analysis usually focuses on specific applications. For example, domain analysis for a product line might identify areas of general commonality and variation across a body of relevant functionality (perhaps by examining and comparing legacy systems that have that functionality). Requirements analysis would identify a specific set of commonalities and variations for the family of products to be built.

*What is the relationship between process improvement and product line practice?*

Process improvement is broader in scope. Product line practice focuses on how to use a common set of core assets to modify, assemble, instantiate, or generate multiple products referred to as a product line. The focus is on improvement in product management. Process improvement addresses software engineering in general. While there is overlap between process improvement and product line practice in key practice areas, product line practices provide guidance for one particular approach to software development. However, experience shows that an organization with poorly defined software processes, or lacking the discipline to follow the processes it has defined, will not fare well in the transition to product line practice. Hence, at the very least, product line practice and process improvement need to be engaged hand in hand. As organizations improve their processes, they often enjoy increased productivity. Organizations with greater process maturity can continue to make productivity increases by turning their focus to product line practices.

*Must all products in the software product line share the same architecture? If my products have different architectures but make heavy use of other shared assets, isn't that a software product line?*

Only if that reuse is carried out in a "prescribed way," as required by the definition of a software product line. In all the software product lines we have studied, that prescription is most effectively carried out by using a common architecture where individual products either share the same architecture or permitted variations of the same architecture—an architecture we call the *product line architecture*. Variations might, for example, involve replacing one component with a similar one, instantiating a multiple component a different number of times, or exhibiting some subset of the overall architecture.

## Using Software Product Lines with Other Approaches

*Can I use open source software packages in my product line?*

You can if all of the following conditions exist:

- You can live with not having any control over the release schedule of the open source package.
- The provided variation mechanisms of the open source package are appropriate for your need to integrate the package into your product line. Most open source packages have variation mechanisms built in to support the disparate needs of a broad user community. If the adaptations you have to make to integrate the package into your product line can be handled by those provided mechanisms, you'll be in good shape when you have to integrate a new release of the package. If you have to make more substantial changes in the package, be prepared to redo all of those changes with every new release. Resist the temptation to clone and own! Otherwise, you'll lose the benefits of having a user community.
- The maturity of the package is appropriate for your needs. There are very robust packages available but also an enormous number of alpha or beta releases. Some of those products are also abandoned and never reach a mature state. When selecting a package, do some research about the popularity of that package and find out whether an active user community exists. If it does, you most likely will get a package at least as robust as what you would expect from your own in-house development group.

Also be aware that some open source licenses are crafted in such a way that would require you to make your products, when based on open source packages, also open source. Read the license agreements very carefully!

*Does the software product line approach work in a globalization environment?*

*Globalization* has two meanings. The first, also called *internationalization* or *localization*, refers to making a software product or software-intensive system work correctly around the world. You can easily see that the software product line approach applies straightforwardly in this case. Different locale requirements correspond directly to product line variation points, and, in fact, a scoping exercise and a product line perspective on requirements can greatly help in identifying those variations. Identifying variations can result in each member of the family being as lean as possible and perhaps letting developers concentrate more on global features as well as locale-specific features.

The second meaning of globalization involves separate development groups located around the world cooperating productively and correctly to build software. In this context, the question above asks whether product lines can be developed globally. They can. It is necessary to lay some groundwork before effective distributed development can occur, and product lines are not immune from this need. For example, it is very useful to first establish such things as a common development environment, a common configuration management system, and a minimum set of common processes (or at least crisp process interface points), so the exchange of information and artifacts can flow smoothly across site boundaries. Just as in global development for single systems, architecture plays a central role. It crisply defines the work assignments and responsibility boundaries, in the system but also among the development parties. In a product line situation, the responsibilities will include the design and development of variation mechanisms for core assets.

*Can a product line approach be compatible with agile development methods?*

The short answer is yes, as demonstrated by the successful use of eXtreme Programming (XP) in Salion's product line effort [Clements 2002d]. However, the larger point is that the applicability of agile methods is more strongly determined by whether a project's characteristics align with a method's "home ground." (See the example practices under the "Process Discipline" practice area.)

Boehm and Turner advocate a pragmatic, risk-driven approach to choosing appropriate aspects from both plan-driven and agile methods [Boehm 2004b]. For projects whose characteristics stray from agility's home ground, it may still be possible to partition off portions where agile methods can flourish.

One challenge to agile methods' applicability is the principle of simple design, de-emphasizing the importance of software architecture. Within XP, this concept is known as "You Aren't Going to Need It" (YAGNI). As Boehm says, YAGNI works fine when future requirements are largely unpredictable but can be highly inefficient where there is a reasonable understanding of future needs. Because a product line approach inherently means you set out to understand future needs, and indeed base your business strategy on that understanding, you *are* going to need a sufficiently defined product line architecture. However, once it's developed, the software architecture contributes very well to a team's tacit knowledge and can serve as a basis for other agile practices (e.g., for development activities within a partitioned area of the architecture).

*Is a model-driven development (MDD) approach compatible with the software product line strategy?*

Yes. The software product line strategy is compatible with a variety of technical approaches loosely grouped under the "model-driven" category including MDD, model-driven architecture, and model-driven engineering. The software engineering practice areas in the framework include the activities of a traditional development process but do not constrain how those activities are implemented. The model-driven techniques emphasize the disciplined use of a variety of models, whose currency is maintained throughout the product life cycle. A model-driven approach to software products lines would emphasize automatic product derivation [McGregor 2005b]. Models would be constructed that are capable of expressing the commonality and variations inherent in the product line's scope. A new product's requirements would be translated into product-specific configurations of the product independent models that are maintained as core assets. Tool-supported translation, laid out in a production plan, would be used to derive products. In fact, model-driven development could be thought of as a production strategy, as described in "Core Asset Development" of the framework.

*Does a system-of-systems (SoS) context rule out the use of software product lines?*

No. On the contrary, software product lines can help reduce the complexity of an SoS context. An SoS comprises independent, self-contained systems that, when taken as a whole, satisfy a specified need. Many software-intensive contexts today are SoSs. The complexity involved can be daunting. However, in many cases, there is considerable commonality among some of the self-contained systems. Suppose, for example, that the SoS involves 200 separate systems that must all interoperate. But suppose further (as is often the case) that there are some clusters within those 200 that have considerable commonality and whose variations could be handled economically. By making those clusters into software product lines, you can tame the interoperability issue into a smaller, more manageable set of interfaces and thereby reduce the complexity of the SoS. Moreover, the economic advantages and predictability associated with software product lines will be highly beneficial to the SoS in question.

*Is there any connection between service-oriented architectures (SOAs) and software product lines?*

SOAs and software product line approaches to software development share a common goal. They both encourage an organization to reuse existing assets and capabilities rather than repeatedly redeveloping them for new systems to achieve desired benefits such as productivity gains, decreased development costs, improved time to market, higher reliability, and competitive advantage.

*How are SOA and software product line approaches alike?*

Both approaches promote reuse by developing applications/products based on a set of reusable components. Those components are developed with well-defined interfaces and processes that specify how the components are to be used, which enables applications/products to be produced in less time.

Adopting either approach requires implementing similar organizational policies and practices necessary to adopt a new technology or a new way of doing business. SOA and software product lines share many of the same organizational issues such as planning, funding, tool support, training, and the need to change the organizational mindset towards reuse. When starting to use either approach, it is imperative for both SOA and software product line efforts to have organizational commitment and a champion.

Both approaches focus on identifying the application building blocks or reusable components associated with the application(s). In SOA, services represent the reusable building blocks. Core assets are the basis for production of products in a software product line. Separate teams may be employed to develop the reusable components and the applications. Small or pilot studies help develop skills to evolve the organization towards an SOA or software product line capability.

In both cases, the initial building blocks may come from legacy systems. Identifying and retrieving product line assets or services from existing systems in order to obtain the benefits of reuse are equally difficult. Documentation and tool support aid this effort.

Application/product development for both approaches is orchestrated in a similar manner. Inputs include the requirements for a particular application/product, reusable components, and the details of how these components are to be used to build the application/product.

*How are SOA and software product line approaches different?*

While the goals and the use of reusable components in the SOA and software product line approaches are very similar, the process by which the two compose systems are very different.

A software product line is, fundamentally, a set of related products. Each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance, adding any new components that may be necessary, and assembling the collection according to the rules of a common, product-line-wide architecture under the auspices of a production plan. New or updated core assets are rolled back into the core asset base for future systems.

In SOA, it is not necessary for the reusable component(s) to come from a centralized, organization-controlled service base. Multiple organizations may provide the services leading to the possibility that services may change or disappear without notification. While multiple organizations can have responsibility for the core asset base in the software product line paradigm, that is not the usual case.

SOA addresses the issue of variation through orchestration (coordinating the participating services), service versioning, or extensible XML data types (a process of evolving from one format to another without requiring central control of the format).

The product line architecture is a software architecture that satisfies the needs of the products within the product line’s scope. It is a key core asset. SOA is not the software architecture of the system; it is a design philosophy and an approach to software development where

- Services provide reusable functionality with well-defined interfaces.
- An SOA infrastructure enables discovery, composition, and invocation of services.
- Applications are built using functionality from available services.

The product line architecture establishes the quality goals for a system—its performance, reliability, modifiability, and so forth. Since SOA implementations may span enterprise boundaries, the quality attributes are dependent on the Quality of Service (QoS) of each of the included services. Desired end-to-end qualities may be achieved in specifically engineered applications. However, if the services are used in a different context, they may not meet the expected QoS. Service developers need to understand the functional and QoS requirements of potential service users.

In a software product line, an established process for updates to the core asset base is followed as the product line evolves, as more resources become available, as fielded products are maintained, and as technological changes or market shifts affect the product line's scope. Unlike SOA, the core assets in a software product line approach include non-software assets as well as software components. Product line requirements, domain models, test cases, and so on provide significant strategic advantage. Also included in the core asset base is a production plan prescribing how the products are produced from the core assets. SOA employs an SOA governance to facilitate planned reuse of services. SOA governance means the creation, deployment, enforcement, and verification of policies throughout the entire life cycle of SOA artifacts. SOA governance platforms may provide easy service discovery through a centralized service registry and management tools for planned reuse (on the service level), such as tracking subscribers to services, negotiating service level agreements (SLAs), communicating change requests and actual changes in service interfaces and data types.

*Can SOA and software product lines be used together?*

It is possible to build a stand-alone (i.e., non-product-line) application using SOA and to build a software product line without using SOA. In that sense, the two approaches are independent. However, it is also possible to combine the two approaches. In particular, it is possible and feasible for an organization to use services as a reusable core asset with which to build products in a software product line. That is a focus of the "Using Externally Available Software" practice area. Also, service providers and SOA application developers could take a product line approach to the development of services.

*Can the framework provide guidance for a move to SOA?*

Organizations moving to an SOA approach can benefit from software product line information captured in other practice areas as well. The practice areas related to organizational management could help organizations understand important issues in adopting a new reuse-based technology. And those related to technical management and software engineering could help organizations analyze legacy systems, determine make/buy/mine/commission decisions, use existing available software, and understand risk. Documents similar to the product line adoption plan (describing the desired state of the organization and a strategy for achieving that state) and the product line production plan (which prescribes how the products are produced from the reusable components) provide excellent templates for organizations moving to planned, reuse-oriented environments.

Planned and systematic reuse, exploiting economies of scope, and other important software product line issues could feed into the SOA design philosophy to help manage the growth of services and application developments, understand the dependencies between services, and determine the impact of service changes to the applications. The software product line approach would help control complexities created by the combinatorics of services and applications that occur over the entire life cycle of a system.

## Product Lines in the Context of Acquisition

*We're a U.S. government contractor, and our government customer wants to build a product line by buying core assets from us and then staging an open competition to build products using them. How can we possibly compete on the product-building side when all these new companies enter the fray and claim to be able to build products cheaper, because they didn't have to pay for the core assets?*

First, other contractors can't necessarily build derivative products any cheaper. At best, they can probably build them as cheaply, but even that is questionable. In practice, the core asset contractor actually may have an inside track on follow-on product development by virtue of its domain knowledge, expertise, and intimate knowledge of the core assets. Other contractors may have a significant, and potentially steep, learning curve with regard to understanding the functionality and technical characteristics of the core assets, how to appropriately augment them with product-specific assets, and how to integrate and test them to create a finished product. In fact, it is usually those other contractors that are concerned about the unfair advantage the asset contractor has, because, under acquisition reform, contractor experience often plays a large part in the technical evaluation criteria. In fact, in some cases, the core asset contractor is excluded from bidding on follow-on product developments to avoid claims of having an unfair advantage.

What all this boils down to is the need to create a business strategy that effectively balances the interests of the core asset contractor and the acquisition organization (and other prospective product development contractors). The core asset contractor obviously has a vested interest in protecting the competitive edge it has in the marketplace. The core assets it owns are an instantiation of its domain knowledge and software expertise and may be a major factor in maintaining its competitive advantage. Accordingly, there are several options the core asset contractor, in conjunction with the acquisition organization, may want to pursue. They include

- selling the core assets outright to the government (with negotiated government-usage rights) while maintaining all commercial rights to the core assets
- licensing government usage of the core assets on a per-product (or per-asset) licensing basis (e.g., the government pays a license fee for each product developed using the core assets)
- negotiating a follow-on contract with the asset contractor to manage, sustain, and evolve the core assets and provide technical support for product development through the auspices of the acquisition organization
- opening up product development to competition through a leader/follower type of contract with the core asset contractor taking the leader role
- allowing the core asset contractor to compete for product development (the same way as any other contractor) or, alternatively, prohibiting the core asset contractor from competing on product development

- avoiding relying on any one contractor by letting an umbrella contract (competitively) to multiple contractors that allows them to compete and/or collaborate on the follow-on development of individual products using the core asset base; this would allow a greater number of contractors to participate in product development and may open up the door to allow the core asset contractor to compete.
- pursuing one, or a combination, of the above options

While there is no one answer to the question that is posed, the above list (which is non-exhaustive) demonstrates that there are many viable contracting options that can be pursued depending on the goals and acquisition strategy of the government agency and the business strategy of the core asset contractor.

Finally, if the core assets make it so straightforward to produce products based on them, the core asset contractor is in a perfect position to rapidly bring entirely new products to the commercial marketplace, thus taking advantage of the new production capability created courtesy of the government.

*We work for a government contractor that wants to bid on a competition to build products based on core assets developed by another contractor. How can we possibly compete given the intimate knowledge possessed by the asset development contractor?*

Combined with the previous question, this question shows that the grass is always greener on the other side of the fence. The fact that both of these questions are frequently asked suggests that product lines do not particularly tilt the playing field in either direction. In fact, this question is no different than if you had asked “How can I hope to compete, because I didn’t build the commercial off-the-shelf (COTS) software or the government-furnished equipment (GFE) that I’m required to use under this contract?” Contractors compete successfully under those conditions all the time. So the premise behind the question—that a contractor can only compete successfully if it is tasked with building *all* the software—is just not valid. A pragmatic answer is that your company is no worse off than if the product line strategy had not been pursued.

*I’m a government contractor, and the government wants me to supply reusable core components for other organizations to use in building a product line. Why should I open myself up to the legal liability? I’ll be liable if they use my components incorrectly!*

Liability issues can be tricky. They are negotiated between the acquisition organization and the development contractor(s) and often involve legal counsel. However, in the case cited, the government contractor would not be directly liable for how another (third-party) government contractor uses its product, especially if it uses the product incorrectly. A contractor who develops components for a government agency is liable to the government—not to another government contractor—to the extent stipulated in the expressed warranties that are part of the contract. There may also be implied warranties of fitness for use for the particular purpose for which the government will use the items. Contracting officers have to consult with counsel prior to asserting any claim for breach of an implied warranty.

The important thing to remember is that liability issues are not unique to product lines. The government often contracts for a piece of equipment from one manufacturer and then provides it as government-furnished equipment (GFE) to another contractor to integrate into its final product. In such cases, the liability for the GFE items rest with the government. However, the government may have recourse to go back to the original development contractor to have a defect corrected, depending on the particular contractual warranties that were negotiated and agreed to by both parties.

For commercial items (i.e., nondevelopmental items), liability considerations (expressed and implied) are described in Part 12 (Acquisition of Commercial Items) of the Federal Acquisition Regulations [FAR 2005a]. General and specific liability considerations that apply to both commercial items and developmental items are described in Part 46 (Quality Assurance) of the FAR.

Specific solicitation provisions and contract clauses on warranties and liabilities that may apply are described in Part 52 of the FAR and Part 252 of the of the Defense Federal Acquisition Regulation Supplement [DFARS 1998a] and include the following sections:

- FAR: 52.246-18; Warranty of Supplies of a Complex Nature
- FAR: 52.266-19; Warranty of Systems and Equipment Under Performance Specifications or Design Criteria
- FAR: 52.246-23; Limitation of Liability
- FAR: 52.246-24; Limitation of Liability-High Value Items
- DFARS: 252.246.7001; Data Warranty

The bottom line is that, during the contract solicitation period, any contractor considering bidding on a government contract can formally request that the contracting officer define (in writing) the extent and limitation of the contractor's liabilities.

*What acquisition strategies should a DoD or government organization consider when acquiring a product line? And what is their potential impact on the acquiring organization?*

There are several alternative acquisition approaches that a program manager should consider when contemplating adopting a product line approach. Three such approaches are

- Commission a government organization to develop the product line. This strategy involves acquiring a completely government-owned product line using the in-house capabilities of a designated government acquisition organization.
- Commission a supplier to develop the product line. This strategy involves acquiring a complete product line production capability and developing derivative products through contracting with one or more suppliers.
- Commission a supplier to develop products using its own product line. This strategy involves acquiring products directly from a supplier who has an existing product line and a demonstrated capability to build derivative products.

The potential impact of these particular approaches on the acquisition organization is summarized in Table 9.

*Table 9: Impact of Acquisition Strategies*

| Product Line Acquisition Approach | Relative degree of organizational sophistication needed by acquirer | Relative degree of acquisition complexity |
|---|---|---|
| 1a. Development by acquisition organization | HIGH | LOW |
| 1.b Development by acquisition organization and later transitioned to contractor | HIGH | MEDIUM |
| 2.a Development involves one supplier | HIGH | HIGH |
| 2.b Development involves multiple suppliers | HIGH+++ | HIGH+++ |
| 3.a Single product acquired from supplier-owned product line | LOW | LOW |
| 3.b Multiple products acquired from supplier-owned product line | LOW | MEDIUM |

Bergey and Cohen describe these strategies in more detail and cite example product line applications [Bergey 2006a].

## Getting Started

*All right, I want to start a software product line. What do I do first?*

Begin by learning. There are many resources available to you, many of which are available on or through the SEI's Web site. You can

- Read case studies and experience reports of organizations pursuing software product lines. They will help you lay out some specific goals for adopting a product line approach and gain an understanding of what might be involved in your particular situation. Assign the most applicable one(s) as reading material for interested people in your organization.
- Get to know the SEI *Framework for Software Product Line Practice,*$^{SM}$ especially essential activities.
- Determine an approach for adopting software product line practices in your organization. You'll need to choose an adoption strategy. The "Launching and Institutionalizing" practice area lists several approaches. You can also use product line practice patterns, described in Chapter 7 of Software Product Lines: Practices and Patterns [Clements 2002c], in particular, the Adoption Factory Pattern [Northrop 2004a], to help with adoption and getting the product line started.
- Take one or more product line courses, such as those in the SEI Software Product Line curriculum.
- Start to build a business case for making the switch to software product lines.
- Get involved with the software product line community by participating in one of the conferences and workshops.

Once you've done some of this suggested homework, you can take the following steps in your organization:

- Conduct a product line diagnosis to help gauge your organization’s strengths and weaknesses with respect to software product line capability. We recommend that you arrange for an SEI Product Line Quick Look or an SEI Product Line Technical Probe. Because the probe relies on interviews with many different people throughout the organization, it’s also a good activity for getting people to think about the product line approach and the organization’s goals for adopting it.
- Build a product line adoption plan for your organization. The Adoption Factory Pattern can be very helpful [Northrop 2004a]. The SEI Adopting Software Product Lines course provides useful guidance and support materials.
- Use the What to Build pattern to help lay out the scope and business case for your software product line [Clements 2002c].

Other steps are involved, but the ones listed above will get you off to a good start.

*My company builds a broadly related group of products, each of which is a group of closely related products. Should I plan to build a single product line or a group of product lines?*

This question can be answered by making a business case for each alternative and weighing the costs and benefits. The SEI Structure Intuitive Model for Product Line Economics (SIMPLE) can help with this [Clements 2005a]. We’ve seen this situation in practice many times. Often, the choice is to go with the single, large product line. Examples come from avionics, missile software, embedded engine controllers, and shipboard command and control systems. However, other organizations have chosen separate product lines or even hierarchical product lines. A hierarchical product line is essentially a product line of product lines. The decision depends on the amount of commonality that can be extracted from the broadly related group, how expensive it is to accommodate the needed variation, and how easy it is to communicate and cooperate across the different groups involved. If the products “live” in separate business units, for instance, the organization might find separate product lines to be more manageable. If you’re just beginning to use the product line concept, it will be much easier to launch one than several, but having a scope that is too broad will jeopardize success with the product line approach.

*I want to pilot software product lines in my organization. What are the criteria for a good pilot project?*

The general criteria are the same as for any pilot project. It should be manageable but not too complex. It should be strategic but not so central that the failure of the pilot will bring down the organization. It should build on strengths rather than weaknesses. Specific to product lines, a pilot should be in an established and well-understood problem area and be led by some of the best innovators. Starting with a legacy system rather than starting from scratch makes sense if the legacy system is in good health (that is, if it’s architecturally sound, well documented, and uses modern technologies). If an established core asset base is available to jump-start the product line, so much the better. Finally, the scope of the pilot should be narrow enough for results to be achieved and assessed quickly. See the “Launching and Institutionalizing” practice area for more information.

*Should I plan to take the proactive approach or the reactive approach to my software product line?*

As described in “All Three Together,” the proactive approach involves building the core assets first and then using them to spin out products, whereas the reactive approach calls for having products first and then extracting the core assets from them. These approaches are two extremes of a spectrum of possibilities in between. A likely compromise is to develop some core assets fresh while producing others from an existing product or stable of products. Key determinants are how well you can predict the future and your available time and resources. If you can, with high confidence, map out the scope of your software product line in detail (describing the commonalities and variations that your products will require), proactively building the core assets to serve that scope will probably pay off in terms of a rapid product-production capability. If, on the other hand, your scope is defined only in more general terms, trying to build the core assets ahead of time will probably result in a large amount of rework and frustration. In that case, it is better to refine the core assets as you field more products and gain more knowledge about your market. The rule of thumb is to use whatever information you have when you have it.

*Where is the resistance to adopting a product line approach usually found?*

Although it can come from almost anywhere, it often shows up at the middle-management level. Many times, the folks in the trenches recognize the benefits of doing things the new way, because they’re the ones tasked with implementing and re-implementing almost identical applications multiple times. Conversely, senior management tends to have the vision and see the financial bottom lines. That is by no means always the case, however. We also know of an organization in which senior management was preoccupied with buyouts and mergers, and middle management stepped in and championed the transition to product line practice. When senior management turned their attention inward, they discovered to their surprise that their company was building software in an entirely new way.

*Where can I go to get more information?*

The SEI’s Web site on product line practice (http://www.sei.cmu.edu/productlines) is a good place to start. There are several books on software product lines, including *Software Product Lines: Practices and Patterns* [Clements 2002c], *Software Product-Line Engineering: A Family-Based Software Development Process* [Weiss 1999a], *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach* [Bosch 2000a], and *Software Product Lines in Action* [van der Linden 2007a]. In addition, *Software Reuse: Architecture, Process, and Organization for Business Success* [Jacobson 1997a] is oriented towards projects employing large-scale strategic reuse and is compatible in many ways with product line practice.

Conferences on software product lines are also emerging, leaving papers about both theory and practice in their wake. The flagship gathering for the field is the International Software Product Line Conference.

*Where can I read about other organizations that have successfully adopted the software product line approach?*

Case studies are excellent resources to help you get started, because they show you how other organizations tackled the product line issues they faced. Case studies are listed on the SEI’s Web site. In addition, you should visit the Software Product Line Hall of Fame, where software product lines of lasting community value are inducted at each Software Product Line Conference. Each inducted product line is described, and references for more information are given. Finally, some of the product line books mentioned above include several case studies of successful product lines, especially *Software Product Lines: Practices and Patterns* [Clements 2002c] and *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering* [van der Linden 2007a].

## Glossary

| | |
|---|---|
| **acquisition** | The process of obtaining products and services via a contract or license. |
| **acquisition strategy** | A plan of action for achieving a specific goal or result through contracting or licensing for products and services. |
| **architectural view** | A representation of a set of system elements and the relationships among them. |
| **attached process** | The process associated with a core asset that tells a product builder how the core asset will be used in the development of products. |
| **business case** | A tool that helps one you make business decisions by predicting how they will affect an organization. Business cases are used among other things to determine if pursuing a product line approach will be beneficial and to determine if a given product line scope makes business sense. |
| **commission** | To contract with another party to build a product or provide a service. |
| **component** | A unit of software composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Szyperski 1998a]. |
| **concept of operations** | Description of an organization's structure, roles, responsibilities, communication mechanisms, processes, practices, and policies that all detail the way the organization operates. |
| **configuration management** | A discipline for evaluating, coordinating, approving or disapproving, and implementing changes in the artifacts that are used to construct and maintain software systems. An artifact can be a piece of hardware or software or documentation. |
| **core asset** | A reusable artifact or resource that is used in the production of more than one product in a software product line. A core asset may be an architecture, a software component, a domain model, a requirements statement or specification, a document, a plan, a test case, a process description, or any other useful element of a software production process. |
| **core asset base** | The complete set of core assets associated with a given software product line. |
| **customer interface** | The description of an organization's connection to its customer(s) including the people involved, the information flow, the communication content, and any applicable policies and procedures. |
| **development** | A generic word used to describe how software comes to be. |
| **domain** | An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area. |

| | |
|---|---|
| **domain analysis** | A process for capturing and representing information about applications in a domain, specifically common characteristics, variations, and reasons for variation. |
| **domain understanding** | Extensive insight and experience in the domains relevant to an organization’s software and/or system endeavors. |
| **externally available software** | Existing software that can be used free, licensed, or purchased. The options for externally available software include commercial off-the-shelf (COTS) software, open source software, freeware, and Web-based services. |
| **Framework for Software Product Line Practice** | A product line encyclopedia that describes the essential activities and practices in which an organization must be competent in order to reap the maximum benefit from fielding a soft-ware product line. The framework was developed and is maintained by the SEI. |
| **market analysis** | The systematic research and analysis of the external factors that determine the success of a product in the marketplace. |
| **mining** | Finding, analyzing, and rehabilitating a piece of an existing software system to serve in a new system for which it was not originally intended. |
| **organizational management practice areas** | Those practice areas necessary for orchestrating the entire software product line effort. |
| **platform** | A word some use to mean the software assets in a product line core asset base. |
| **practice area** | A body of work or a collection of activities that an organization must master to successfully carry out the essential work of a software product line. |
| **product** | Deployed software-intensive system or software. |
| **product constraints** | The set of common and variant features and behavioral attributes associated with the products in the product line scope. |
| **product line** | A set of products that share a common, managed set of features satisfying the needs of a particular market segment. |
| **product line adoption** | An organization’s change to a software product line approach, which involves developing a core asset base, supportive processes, and organizational structures; developing products from that asset base in a way that achieves business goals; and preparing itself to institutionalize product line practices. |
| **product line adoption plan** | An organizational plan that describes how product line practices will be rolled out across the organization. |
| **product line approach** | The technical and business practices necessary to build a family of products as a software product line. |
| **product line architecture** | A core asset that is the software architecture for all the products in a software product line. A product line architecture explicitly provides variation mechanisms that support the diversity among the products in the software product line. |
| **product line scope** | A description of the products that will constitute the product line or that the product line is capable of including. |

| | |
|---|---|
| **production plan** | The guide to how products in the software product line will be constructed from the product line's core assets. |
| **production constraints** | Any restrictions on the timing, development environment, processes, or developer skills associated with development of the products in a software product line. |
| **production capability** | The core asset base, supportive processes, and tools that enable the development of the products in a software product line. |
| **production method** | The overall implementation approach that specifies the models, processes, and tools used in the attached processes across core assets. |
| **production process** | The process used for building all products in a software product line. The production process is defined by the set of attached processes with the necessary process "glue" to join them together into a coherent whole. |
| **production strategy** | The overall approach for realizing both the core assets and products in a software product line. |
| **project** | A temporary endeavor aimed at creating a unique product or service. Typically a project has its own funding, accounting, and delivery schedule. |
| **requirements engineering** | The use of systematic and repeatable techniques to elicit, analyze, specify, verify, and manage system requirements. |
| **reuse** | Using an item more than once. |
| **scoping** | An activity that bounds the behaviors and features of a system or set of systems. In a product line approach, scoping is the activity that defines the product line scope. |
| **software architecture** | Structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 2003a]. |
| **software engineering practice areas** | Those practice areas necessary for applying the appropriate technology to create and evolve both core assets and products. |
| **software product line** | A set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. |
| **software product line practice pattern** | A description of an organization's context, the product line problem it is trying to solve, and how a set of practice areas can be used in concert to solve the problem. |
| **strategic reuse** | Planned, systematic reuse that implements tightly connected business and technical strategies. |
| **technical management practice areas** | Those practice areas necessary for managing the creation and evolution of the core assets and the products. |
| **technology forecasting** | Looking at future technologies that will either support internal software development or affect features or capabilities embedded in an organization's products. |

# Bibliography

| | |
|---|---|
| **[Abowd 1996a]** | Abowd, G.; Bass, L.; Clements, P.; Kazman, R.; Northrop, L.; & Zaremski, A. *Recommended Best Industrial Practice for Software Architecture Evaluation* (CMU/SEI-96-TR-025, ADA320786). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Albert 2002a]** | Albert, C.; Brownsword, L.; Bentley, D.; Bono, T.; Morris, E.; & Pruitt, D. *Evolutionary Process for Integrating COTS-Based Systems (EPIC) Building, Fielding, and Supporting Commercial-off-the-Shelf (COTS) Based Solutions* (CMU/SEI-2002-TR-005, ADA408653). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. |
| **[Alexander 1979a]** | Alexander, C. *The Timeless Way of Building*. New York, NY: Oxford University Press, 1979. |
| **[Alhir 2002a]** | Alhir, Sinan Si. “Understanding the Unified Process (UP).” *Methods and Tools 10*, 1 (Spring 2002): 2-17. |
| **[America 2000a]** | America, P.; Obbink, H.; van Ommering, R.; & van der Linden, F. “CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering,” 167-180. *Software Product Lines: Proceedings of the First Software Product Line Conference* (SPLC1). Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Anastasopoulos 2000a]** | Anastasopoulos, M. & Gacek, C. *Implementing Product Line Variabilities* (IESE-Report No. 089.00/E, V1.0). Kaiserslautern, Germany: Fraunhofer Institut Experimentelles Software Engineering, 2000. |
| **[ANSI 1992a]** | American National Standards Institute. *Guide for the Preparation of Operational Concept Documents* (ANSI/AIAA G-043-1992). Washington, DC: American National Standards Institute, 1992. |
| **[AOSA 2007a]** | Aspect-Oriented Software Association. Aspect-Oriented Software Development Home Page (2007). |
| **[Arango 1994a]** | Arango, G. Ch. 2, “Domain Analysis Methods,” 17-49. *Software Reusability*. Hemel Hempstead, England: Ellis Horwood, 1994. |
| **[Ardis 2000a]** | Ardis, M.; Dudak, P.; Dor, L.; Leu, W.; Nakatani, L.; Olsen, B.; & Pontrelli, P. “Domain Engineered Configuration Control,” 479-494. *Software Product Lines: Proceedings of the First Software Product Line Conference* (SPLC1). Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Bachmann 2000a]** | Bachmann, F.; Bass, L.; Chastek, G.; Donohoe, P.; & Peruzzi, F. *The Architecture Based Design Method* (CMU/SEI-2000-TR-001, ADA375851). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. |
| **[Bachmann 2005a]** | Bachmann, F. & Clements, P. *Variability in Software Product Lines* (CMU/SEI-2005-TR-012, ADA450337). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. |
| **[Baldwin 2002a]** | Baldwin, Carliss Y. & Clark, Kim B. “The Option Value of Modularity in Design-An Example from Design Rules, Volume 1: The Power of Modularity” (May 2002). |
| **[Basili 1984a]** | Basili, V. R. & Weiss, D. “A Methodology for Collecting Valid Software Engineering Data.” *IEEE Transactions on Software Engineering SE-10*, 6 (November 1984): 728-738. |
| **[Bass 1997a]** | Bass, L.; Clements, P.; Cohen, S.; Northrop, L.; & Withey, J. *Product Line Practice Workshop Report* (CMU/SEI-97-TR-003, ADA327610). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. |

| | |
|---|---|
| **[Bass 1998a]** | Bass, L.; Chastek, G.; Clements, P.; Northrop, L.; Smith, D.; & Withey, J. *Second Product Line Practice Workshop Report* (CMU/SEI-98-TR-015, ADA354691). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. |
| **[Bass 1999a]** | Bass, L.; Campbell, G.; Clements, P.; Northrop, L.; & Smith, D. *Third Product Line Practice Workshop* (CMU/SEI-99-TR-003, ADA361391). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Bass 1999b]** | Bass, L. & Kazman, R. *Architecture-Based Development* (CMU/SEI-99-TR-007, ADA366100). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Bass 2000a]** | Bass, L.; Clements, P.; Donohoe, P.; McGregor, J.; & Northrop, L. *Fourth Product Line Practice Workshop Report* (CMU/SEI-2000-TR-002, ADA375843). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. |
| **[Bass 2003a]** | Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second Edition*. Reading, MA: Addison-Wesley, 2003. |
| **[Batory 2005a]** | Batory, Don. “Feature Models, Grammars, and Propositional Formulas,” 7-20. *Proceedings of the 9th International Software Product Line Conference (SPLC 2005)* (Lecture Notes in Computer Science volume 3714). Rennes, France, September 26-29, 2005. New York, NY: Springer, 2005. |
| **[Bayer 2000a]** | Bayer, J.; Muthig, D.; & Widen, T. “Customizable Domain Analysis,” 178-194. *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE ‘99)*. Erfurt, Germany, September 28-30, 1999. New York, NY: Springer, 2000. |
| **[Bean 2000a]** | Bean, J. “Use XML Even As It Changes.” *Enterprise Development 2*, 2 (February 2000): 44-50. |
| **[Beck 1994a]** | Beck, K. & Johnson, R. “Patterns Generate Architectures,” 139-149. *Proceedings of the Eighth European Conference on Object-Oriented Programming (ECOOP ‘94)*. Bologna, Italy, July 4-8, 1994. New York, NY: Springer-Verlag, 1994. |
| **[Beck 1999a]** | Beck, K. *Extreme Programming Explained*. Reading, MA: Addison-Wesley, 1999. |
| **[Beck 2002a]** | Beck, K. *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2002. |
| **[Beizer 1990a]** | Beizer, B. *Software Testing Techniques*. Boston, MA: International Thompson Computer Press, 1990. |
| **[Berczuk 2003a]** | Berczuk, Steve. *Software Configuration Management Patterns*. Boston, MA: Addison-Wesley, 2003. |
| **[Bergey 1998a]** | Bergey, J.; Clements, P.; Cohen, S.; Donohoe, P.; Jones, L.; Krut, B.; Northrop, L.; Tilley, S.; Smith, D.; & Withey, J. *DoD Product Line Practice Workshop Report* (CMU/SEI-98-TR-007, ADA346252). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. |
| **[Bergey 1999a]** | Bergey, J.; Smith, D.; Weiderman, N.; & Woods, S. *Options Analysis for Reengineering (OAR): Issues and Conceptual Approach* (CMU/SEI-99-TN-014, ADA370600). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Bergey 1999b]** | Bergey, J.; Fisher, M.; & Jones, L. *The DoD Acquisition Environment and Software Product Lines* (CMU/SEI-99-TN-004, ADA244787). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |

| | |
|---|---|
| **[Bergey 2001a]** | Bergey, J.; O'Brien, L.; & Smith, D. *Options Analysis for Reengineering (OAR): A Method for Mining Legacy Assets* (CMU/SEI-2001-TN-013, ADA395201). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. |
| **[Bergey 2002a]** | Bergey, J.; O'Brien, L.; & Smith, D. "Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line," 316-327. *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*. San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002. |
| **[Bergey 2003a]** | Bergey, J.; O'Brien, L.; & Smith, D. *Application of Options Analysis for Reengineering in a Lead System Integrator Environment* (CMU/SEI-2003-TN-009). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. |
| **[Bergey 2004a]** | Bergey, J.; Campbell, G.; Cohen, S.; Fisher, M.; Gallagher, B.; Jones, L.; Northrop, L.; & Soule, A. *Software Product Line Acquisition: A Companion to a Framework for Software Product Line Practice, Version 3.0*. (2004). |
| **[Bergey 2006a]** | Bergey, John & Cohen, Sholom. *Product Line Acquisition in a DoD Organization-Guidance for Decision Makers* (CMU/SEI-2006-TN-020, ADA447911). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006. |
| **[Birk 2003a]** | Birk, A.; Heller, G.; John, I.; Joos, S.; Muller, K.; Schmid, K.; & von der Massen, T. *Report of the GI Work Group "Requirements Engineering for Product Lines"* (IESE-Report No. 121.03/E, V1.0). Kaiserslautern, Germany: Fraunhofer Institut Experimentelles Software Engineering, 2003. |
| **[Boeckle 2002a]** | Boeckle, G.; Munoz, J.; Knauber, P.; Krueger, C.; Leite, J.; van der Linden, F.; Northrop, L.; Stark, M.; & Weiss, D. "Adopting and Institutionalizing a Product Line Culture,"49-59. *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*. San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002. |
| **[Boehm 1981a]** | Boehm, B. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981. |
| **[Boehm 1988a]** | Boehm, B. W. "A Spiral Model of Software Development and Enhancement." *Computer 21*, 5 (May 1988): 61-72. |
| **[Boehm 1989a]** | Boehm, B. *IEEE Tutorial on Software Risk Management*. Piscataway, NJ: IEEE Computer Society Press, 1989. |
| **[Boehm 2000a]** | Boehm, B. "…And Very Few Lead Bullets, Either" [CD-ROM]. *Proceedings of Impacts 2000: The 15th Annual Software Engineering Symposium*. Washington, DC, September 18-21, 2000. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. |
| **[Boehm 2004a]** | Boehm, Barry; Brown, A. Winsor; Madachy, Ray; & Yang, Ye. "A Software Product Line Life Cycle Cost Estimation Model," 156-164. *Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2004)*. Redondo Beach, CA, August 19-20, 2004. Los Alamitos, CA: IEEE Computer Society, 2004. |
| **[Boehm 2004b]** | Boehm, B. & Turner, R. *Balancing Agility and Discipline: A Guide for the Perplexed*. Reading, MA: Addison-Wesley, 2004. |
| **[Booch 1994a]** | Booch, G. *Object-Oriented Analysis and Design with Applications*. Reading, MA: Addison-Wesley, 1994. |
| **[Bosch 2000a]** | Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Reading, MA: Addison-Wesley, 2000. |

| | |
|---|---|
| **[Bosch 2000b]** | Bosch, J. “Organizing for Software Product Lines,” 117-134. *Proceedings of the 3rd International Workshop on Software Architectures for Product Families (IWSAPF-3)*. Las Palmas de Gran Canaria, Spain, March 15-17, 2000. Berlin, Germany: Springer, 2000. |
| **[Bosch 2002a]** | Bosch, J. “Maturity and Evolution in Software Product Lines: Approaches, Artifacts, and Organization,” 257-271. *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*. San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002. |
| **[Brassard 2001a]** | Brassard, M. & Ritter, D. *Sailing Through Six Sigma*. Marietta, GA: Brassard & Ritter, 2001. |
| **[Brooks 1987a]** | Brooks, F. “No Silver Bullet: Essence and Accidents of Software Engineering.” *Computer 20*, 4 (April 1987): 10-19. |
| **[Brown 1994a]** | Brown, A. W.; Carney, D. J.; Morris, E. J.; Smith, D. B.; & Zarrella, P. F. *Principles of Case Tool Integration*. Oxford, U.K.: Oxford University Press, 1994. |
| **[Brown 1994b]** | Brown, A. W. “Why Evaluating CASE Environments is Different from Evaluating CASE Tools,” 4-13. *Proceedings of the Third Symposium on Assessment of Quality Software Development Tools*. Washington, DC, June 7-9, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994. |
| **[Brown 1996a]** | Brown, A. & Wallnau, K. “A Framework for Evaluating Software Technology.” *IEEE Software 13*, 5 (September 1996): 39-49. |
| **[Brown 1998a]** | Brown, A. W. & Wallnau, K. C. “The Current State of CBSE.” *IEEE Software 15*, 5 (September/October 1998): 37-46. |
| **[Brownsword 1996a]** | Brownsword, L. & Clements, P. *A Case Study in Successful Product Line Development* (CMU/SEI-96-TR-016, ADA315802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Bruckhaus 1996a]** | Bruckhaus, T.; Madhavji, N. H.; Janssen, I.; & Henshaw, J. “The Impact of Tools on Software Productivity.” *IEEE Software 13*, 5 (September 1996): 29-38. |
| **[Budgen 2003a]** | Budgen, David & Thomson, Mitchell. “CASE Tool Evaluation: Experiences from an Empirical Study.” *The Journal of Systems and Software 67*, 2 (2003): 55-75. |
| **[Burrows 2005a]** | Burrows, Clive & Wesley, Ian. *Ovum Evaluates: Configuration Management*. London, UK: Ovum, Ltd., 2005. |
| **[Buschmann 1996a]** | Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY: John Wiley & Sons, 1996. |
| **[Cagan 1992a]** | Cagan, M. & Wright, A. *Requirements for a Modern Software Configuration Management System*. Irvine, CA: Continuous Software Corporation, currently Telelogic, Inc., 1992. |
| **[CARDS 1994a]** | Comprehensive Approach to Reusable Defense Software (CARDS). *Training Plan* (STARS-VC-B003/001/00). Reston, VA: Unisys Corporation, 1994. |
| **[Carney 1997a]** | Carney, D. *Assembling Large Systems from COTS Components: Opportunities, Cautions, and Complexities*. SEI Monographs on the Use of Commercial Software in Government Systems. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. |
| **[Carney 1998a]** | Carney, D.; Brownsword, L.; & Oberndorf, T. “The Opportunities and Complexities of Applying Commercial Off-the-Shelf Components.” *Crosstalk 11*, 4 (April 1998): 4-6. |

| | |
|---|---|
| **[Carney 1998b]** | Carney, D. “Evaluation of COTS Products: Some Thoughts on the Process” [online]. *SEI Interactive 1*, 2 (September 1998). |
| **[Carney 1998c]** | Carney, D. “COTS Evaluation in the Real World” [online]. *SEI Interactive 1*, 3 (December 1998). |
| **[Charan 1999a]** | Charan, R. & Colvin, G. “Why CEOs Fail.” *Fortune 139*, 12 (June 21, 1999): 68-78. |
| **[Charette 1989a]** | Charette, R. *Software Engineering Risk Analysis and Management*. New York, NY: McGraw-Hill, 1989. |
| **[Chastek 2001a]** | Chastek, G.; Donohoe, P.; Kang, K.; & Thiel, S. *Product Line Analysis: A Practical Introduction* (CMU/SEI-2001-TR-001, ADA396137). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. |
| **[Chastek 2002a]** | Chastek, G., ed. *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*. San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002. |
| **[Chastek 2002b]** | Chastek, G. & McGregor, J. *Guidelines for Developing a Product Line Production Plan* (CMU/SEI-2002-TR-006, ADA407772). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. |
| **[Chastek 2002c]** | Chastek, Gary; Donohoe, Patrick; & McGregor, John D. *Product Line Production Planning for the Home Integration System Example* (CMU/SEI-2002-TN-029, ADA405846). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. |
| **[Chastek 2003a]** | Chastek, Gary & Donohoe, Patrick. *Product Line Analysis for Practitioners* (CMU/SEI-2003-TR-008). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. |
| **[Chastek 2004a]** | Chastek, Gary; Donohoe, Patrick; & McGregor, John D. *A Study of Product Production in Software Product Lines* (CMU/SEI-2004-TN-012), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. |
| **[Christensen 1997a]** | Christensen, Clayton M. *The Innovator’s Dilemma: When New Technologies Cause Great Firms to Fail*. Boston, MA: Harvard Business School Press, 1997. |
| **[Clemen 1991a]** | Clemen, R. T. *Making Hard Decisions: An Introduction to Decision Analysis*. Boston, MA: PWS-Kent Publishing Co., 1991. |
| **[Clements 1998a]** | Clements, P.; Bass, L.; Chastek, G.; Northrop, L.; Smith, D.; & Withey, J. *Second Product Line Practice Workshop Report* (CMU/SEI-98-TR-015, ADA354691). Colorado Springs, CO, November 1997. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. |
| **[Clements 1998b]** | Clements, P. & Weiderman, N. *Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families* (CMU/SEI-98-SR-003, ADA346343). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. |
| **[Clements 2000a]** | Clements, P. *Active Reviews for Intermediate Designs* (CMU/SEI-2000-TN-009, ADA383775). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. |
| **[Clements 2001a]** | Clements, P.; Kazman, R.; & Klein, M. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison-Wesley, 2001. |
| **[Clements 2002a]** | Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; & Stafford, J. *Documenting Software Architectures: Views and Beyond*. Reading, MA: Addison-Wesley, 2002. |

| | |
|---|---|
| **[Clements 2002b]** | Clements, P. & Krueger, C. Two-part Point/Counterpoint column: “Being Proactive Pays Off” and “Eliminating the Adoption Barrier.” *IEEE Software 19*, 4 (July/August 2002): 28-31. |
| **[Clements 2002c]** | Clements, P. & Northrop, L. *Software Product Lines: Practices and Patterns*. Boston, MA: Addison-Wesley, 2002. |
| **[Clements 2005a]** | Clements, Paul C.; McGregor, John D.; & Cohen, Sholom G. *The Structured Intuitive Model for Product Line Economics (SIMPLE)* (CMU/SEI-2005-TR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. |
| **[Clements 2005b]** | Clements, P.; Jones, L.; McGregor, J.; & Northrop, L. “Project Management in a Software Product Line Organization.” *IEEE Software 22*, 5 (September/October 2005): 54-62. |
| **[Clemons 1997a]** | Clemons, Eric K. & Hitt, Lorin M. *Strategic Sourcing for Services: Assessing the Balance Between Outsourcing and Insourcing*. http://opim.wharton.upenn.edu/~clemons/files/outsourcing_v4_2.pdf (June 1997). |
| **[Cohen 1991a]** | Cohen, S. G.; Stanley Jr., J. L.; Peterson, A. S.; & Krut Jr., R. W. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain and Appendices A-I* (CMU/SEI-91-TR-028, ADA256590). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1991. |
| **[Cohen 1996a]** | Cohen, S.; Friedman, S.; Martin, L.; Royer, T.; Solderitsch, N.; & Webster, R. *Concept of Operations for the ESC Product Line Approach* (CMU/SEI-96-TR-018, ADA313952). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Cohen 1998a]** | Cohen, S. & Northrop, L. “Object-Oriented Technology and Domain Analysis,” 86-93. *Proceedings of the Fifth International Conference on Software Reuse*. Victoria, B.C., June 2-5, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998. |
| **[Cohen 1999a]** | Cohen, S. *Guidelines for Developing a Product Line Concept of Operations* (CMU/SEI-99-TR-008, ADA367714). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Cohen 2001a]** | Cohen, S. *Case Study: Building and Communicating a Business Case for a DoD Product Line* (CMU/SEI-2001-TN-020, ADA395155). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. |
| **[Cohen 2003a]** | Cohen, S. *Predicting When Product Line Investment Pays* (CMU/SEI-2003-TN-017, ADA418466). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. |
| **[Coplien 1998a]** | Coplien, J.; Hoffman, D.; & Weiss, D. “Commonality and Variability in Software Engineering.” *IEEE Software 15*, 6 (November/December 1998): 37-45. |
| **[Crosby 1979a]** | Crosby, P. B. *Quality Is Free*. New York, NY: McGraw-Hill, 1979. |
| **[Crossroads 2006a]** | CM Crossroads. CM Crossroads Home Page. http://www.cmcrossroads.com/ (2007). |
| **[Cruikshank 1998a]** | Cruikshank, J. L. & Sicilia, D. B. *The Engine That Could: 75 Years of Value-Driven Change at Cummins Engine Company*. Boston, MA: Harvard Business School Press, 1998. |
| **[Curtis 1992a]** | Curtis, B.; Kellner, M.; & Over, J. “Process Modeling.” *Communications of the ACM 35*, 9 (September 1992): 75-90. |

| | |
|---|---|
| **[Cusumano 1991a]** | Cusumano, M. A. *Japan's Software Factories*. New York, NY: Oxford University Press, 1991. |
| **[Czarnecki 2005a]** | Czarnecki. K. "Overview of Generative Software Development," 326-341. *Unconventional Programming Paradigms (UPP) 2004* (Lecture Notes in Computer Science volume 3566). Le Mont Saint Michel, France, September 15-17, 2004. Berlin, Germany: Springer-Verlag, 2005. |
| **[Dabrowski 1993a]** | Dabrowski, C. & Katz, S. *A Context Analysis of the Network Management Domain* (NISTIR 5309). Gaithersburg, MD: National Institute of Standards and Technology, 1993. |
| **[Dabrowski 1994a]** | Dabrowski, C. & Watkins, J. *A Domain Analysis of the Alarm Surveillance Domain V1.0* (NISTIR 5494). Gaithersburg, MD: National Institute of Standards and Technology, 1994. |
| **[Dager 2000a]** | Dager, J. "Cummins's Experience in Developing a Software Product Line Architecture for Real-Time Embedded Diesel Engine Controls," 23-45. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Dart 1991a]** | Dart, S. "Concepts in Configuration Management Systems," 1-18. *Proceedings of the Third International Conference on Software Configuration Managemen*t. Trondheim, Norway, June 12-14, 1991. New York, NY: Association for Computing Machinery Press, 1991. |
| **[Davis 1990a]** | Davis, A. M. *Software Requirements: Analysis and Specification*. Englewood Cliffs, NJ: Prentice-Hall, 1990. |
| **[DeBaud 1999a]** | DeBaud, J. & Schmid, K. "A Systematic Approach to Derive the Scope of Software Product Lines," 34-43. *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. Los Angeles, CA, May 16-22, 1999. Los Alamitos, CA: IEEE Computer Society, 1999. |
| **[Del Rosso 2006a]** | Del Rosso, C. "Continuous Evolution Through Software Architecture Evaluation: A Case Study." *Journal of Software Maintenance and Evolution Research and Practice 18*, 5 (September/October 2006): 351-383. |
| **[Deming 1986a]** | Deming, W. E. *Out of the Crisis*. Cambridge, MA: MIT Center for Advanced Engineering, 1986. |
| **[Deschamps 1995a]** | Deschamps, J. & Nayak, P. R. *Product Juggernauts*. Watertown, MA: Harvard Business School Press, 1995. |
| **[DFARS 1998a]** | U.S. Department of the Navy. *The Defense Federal Acquisition Regulation Supplement (DFARS)* (1998). |
| **[Diaz 2005a]** | Díaz, Oscar; Trujillo, Salvador; & Anfurrutia, Felipe I. "Supporting Production Strategies as Refinements of the Production Process," 210-221. *Proceedings of the 9th International Software Product Lines Conference (SPLC 2005)*. Rennes, France, September 26-29, 2005. New York, NY: Springer, 2005. |
| **[DISA 1993a]** | Defense Information Systems Agency Center for Information Management (DISA/CIM) Software Reuse Program. *Domain Analysis and Design Process, V1*. Arlington, VA: Defense Information Systems Agency Center for Information Management, 1993. |
| **[DISA 1995a]** | Defense Information Systems Agency. *US Army Space and Strategic Defense Command, Software Reuse Business Model*. Washington, DC: The Department of Defense, 1995. |

| | |
|---|---|
| **[Donohoe 2000a]** | Donohoe, P., ed. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Dorfman 1997a]** | Dorfman, M. & Thayer, R. H. *Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1997. |
| **[Dorofee 1994a]** | Dorofee, A.; Walker, J.; Gluch, D.; Higuera, R.; Murphy, R.; Walker, J.; & Williams, R. *Team Risk Management Guidebook*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994. |
| **[Dorofee 1996a]** | Dorofee, A.; Walker, J.; Alberts, C.; Higuera, R.; Murphy, R.; & Williams, R. *Continuous Risk Management Guidebook*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Dumaine 1989a]** | Dumaine, B. “How Managers Can Succeed by Speed.” *Fortune 119*, 4 (February 1989): 54. |
| **[Ecklund 1996a]** | Ecklund, Jr., E.; Delcambre, L.; & Freiling, M. “Change Cases: Use Cases That Identify Future Requirements,” 342-358. *Conference Proceedings of the OOPSLA 96*. San Jose, CA, October 6-10, 1996. San Jose, CA: ACM Press, 1996. |
| **[Eisner 1994a]** | Eisner, H. “Systems Engineering Sciences,” 1312-1322. *Encyclopedia of Software Engineering, Volume 2*. New York, NY: John Wiley & Sons, 1994. |
| **[Estublier 2005a]** | Estublier, Jacky; Leblang, David; van der Hoek, Andre; Conradi, Reidar; Clemm, Geoffrey; Tichy, Walter; & Wiborg-Weber, Darcy. “Impact of Software Engineering Research on the Practice of Software Configuration Management.” *ACM Transactions on Software Engineering and Methodology 14*, 4 (October 2005): 383-430. |
| **[Etzioni 1964a]** | Etzioni, A. *Modern Organizations*. Englewood Cliffs, NJ: Prentice-Hall, 1964. |
| **[FAA 1995a]** | Federal Aviation Administration (FAA) Office of Information Technology Integrated Product Team for Information Technology Services. Ch. 6, “The Business Case.” *Business Process Improvement (Reengineering) Handbook of Standards and Guidelines*. Washington, DC: Federal Aviation Administration, 1995. |
| **[Fairley 1994a]** | Fairley, R. “Risk Management for Software Projects.” *IEEE Software 2*, 3 (May 1994): 57-67. |
| **[FAR 2005a]** | U.S. Federal Regulation Acquisition Agency. *Federal Acquisition Regulation*. http://www.arnet.gov/far/index.html (2005). |
| **[Faulk 1997a]** | Faulk, S. R. “Software Requirements: A Tutorial,” 128-149. *Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1997. |
| **[Faulk 2000a]** | Faulk, S.; Harmon, R.; & Raffo, D. “Value-Based Software Engineering (VBSE): A Value-Driven Approach to Product-Line Engineering,” 205-224. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Favre 2003a]** | Favre, Jean-Marie; Estublier, Jacky; & Sanlaville, Remy. “Tool Adoption Issues in a Very Large Software Company,” 81-89. *ACSE 2003: Third International Workshop on Adoption-Centric Software Engineering* (CMU/SEI-2003-SR-004, ADA416604). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. |

| | |
|---|---|
| **[Finkelstein 1994a]** | Finkelstein, A.; Kramer, J.; & Nuseibeh, B. *Software Process Modeling and Technology*. New York, NY: John Wiley & Sons, 1994. |
| **[Finnegan 1997a]** | Finnegan, P.; Holt, R.; Kalas, I.; Kerr, S.; Kontogiannis, K.; Muller, H.; Mylopolous, J.; Perelgut, S.; Stanley, M.; & Wong, K. “The Software Bookshelf.” *IBM Systems Journal 36*, 4 (November 1997): 564-593. |
| **[Fowler 1999a]** | Fowler, P.; Middlecoat, B.; & Yo, S. *Lessons Learned Collaborating on a Process for SPI at Xerox* (CMU/SEI-99-TR-006, ADA373332). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Fraunhofer 2006a]** | Fraunhofer Institut Experimentelles Software Engineering (IESE). *Product Line Architectures*. http://www.iese.fraunhofer.de/fhg/Images/CC_PLA_Flyer_e_2006_03_tcm168-62107.pdf (2006). |
| **[Fritsch 2004a]** | Fritsch, C. & Hahn, R. “Product Line Potential Analysis,” 228-237. *Proceedings of the Third Software Product Lines Conference (SPLC 2004)* (Lecture Notes in Computer Science volume 3154). Boston, MA, August 30-September 2, 2004. New York, NY: Springer, 2004. |
| **[FSF 2007a]** | The Free Software Foundation. *The Free Software Foundation Home Page*. http://www.fsf.org/ (2007). |
| **[Gaffney 1992a]** | Gaffney, J. E. & Cruickshank, R. D. “A General Economics Model of Software Reuse,” 327-337. *Proceedings of the 14th International Conference on Software Engineering (ICSE)*. Melbourne, Australia, May 11-15, 1992. New York, NY: ACM, 1992. |
| **[Gallagher 1997a]** | Gallagher, B. P.; Alberts, C. J.; & Barbour R. E. *Software Acquisition Risk Management Key Process Area (KPA)—A Guidebook V1.0* (CMU/SEI-97-HB-002, ADA328098). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. |
| **[Gallagher 1999a]** | Gallagher, B. P. *Software Acquisition Risk Management Key Process Area (KPA)—A Guidebook Version 1.02* (CMU/SEI-99-HB-001, ADA370385). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Gamma 1995a]** | Gamma, E.; Helms, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. |
| **[Ganesan 2005a]** | Ganesan, D. & Knodel, J. “Identifying Domain-Specific Reusable Components from Existing OO Systems to Support Product Line Migration,” 27-36. *R2PL 2005-Proceedings of the First International Workshop on Reengineering Towards Product Lines* (CMU/SEI-2006-SR-002, ADA448167). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. |
| **[Ganesan 2006a]** | Ganesan, D.; Muthig, D.; & Yoshimura, K. “Predicting Return-on-Investment for Product Line Generations,” 13-22. *Proceedings of the 10th International Software Product Lines Conference* (SPLC 2006). Baltimore, MD, August 21-24, 2006. Los Alamitos, CA: IEEE Computer Society, 2006. |
| **[Garlan 1995a]** | Garlan, D.; Allen, R.; & Ockerbloom, J. “Architectural Mismatch: Why Reuse Is So Hard.” *IEEE Software 12*, 6 (November 1995): 17-26. |
| **[Gelenbe 1999a]** | Gelenbe, E, ed. *System Performance Evaluation: Methodologies and Applications*. Boca Raton, FL: CRC Press, 1999. |
| **[Geppert 2003a]** | Geppert, B. & Weiss, D. “Goal-Oriented Assessment of Product-Line Domains,” 180-188. *Proceedings of the Ninth International Software Metrics Symposium (METRICS’03)*. Sydney, Australia, September 3-5, 2003. Los Alamitos, CA: IEEE Computer Society, 2003. |

| | |
|---|---|
| **[Glass 1998a]** | Glass, R. L. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Upper Saddle River, NJ: Prentice-Hall, 1998. |
| **[Gleick 1987a]** | Gleick, J. *Chaos: Making a New Science*. New York, NY: Penguin Books, 1987. |
| **[Goldberg 1995a]** | Goldberg, A. & Rubin, K. S. *Succeeding with Objects: Decision Frameworks for Project Manage-ment*. Reading, MA: Addison-Wesley, 1995. |
| **[Grady 1992a]** | Grady, R. B. *Practical Software Metrics for Project Management and Process Improvement*. Eng-lewood Cliffs, NJ: Prentice-Hall, 1992. |
| **[Grady 1997a]** | Grady, R. B. *Successful Software Process Improvement*. Englewood Cliffs, NJ: Prentice-Hall, 1997. |
| **[Graham 1998a]** | Graham, I. *Requirements Engineering and Rapid Development: An Object-Oriented Approach*. Es-sex, England: Addison-Wesley, 1998. |
| **[Griss 1994a]** | Griss, M. *Software Reuse: Objects and Frameworks Are Not Enough* (HPL-95-03). Palo Alto, CA: Hewlett-Packard, 1994. http://www.hpl.hp.com/techreports/95/HPL-95-03.html |
| **[Griss 1998a]** | Griss, M. L.; Favaro, J.; & d'Alessandro, M. "Integrating Feature Modeling with the RSEB," 76-85. *Proceedings of the Fifth International Conference on Software Reuse*. Victoria, B.C., June 2-5, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998. |
| **[Hammond 1999a]** | Hammond, J. S.; Keeney, R. L.; & Raiffa, H. *Smart Choices: A Practical Guide to Making Better Decisions*. Boston, MA: Harvard Business School Press, 1999. |
| **[Harel 1998a]** | Harel, D. & Politi, M. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. Reading, MA: Addison-Wesley, 1998. |
| **[Hofmeister 2000a]** | Hofmeister, N. S. *Applied Software Architecture*. Reading, MA: Addison-Wesley, 2000. |
| **[Hollander 1999a]** | Hollander, C. R. & Ohlinger, J. "CCT: A Component-Based Product Line Architecture for Satellite-Based Command and Control Systems," 201-206. *Proceedings of the Workshop on Object Tech-nology for Product-Line Architectures*. Lisbon, Portugal, June 15, 1999. Bilbao, Spain: European Software Institute, 1999. |
| **[Hotz 2006a]** | Hotz, L.; Wolter K.; Krebs, T.; Deelstra, S.; Sinnema, M.; Nijhuis, J.; & MacGregor, J. *Configuration in Industrial Product Families - The ConIPF Methodology*. Amsterdam, The Netherlands: IOS Press, 2006. |
| **[Huang 2003a]** | Y. Huang, I. J. Taylor, D. W. Walker, and R. Davies, "Wrapping Legacy Codes for Grid-Based Ap-plications," 139-145. *Proceedings of the 17th International Parallel and Distributed Processing Symposium (Workshop on Java for HPC)*. Nice, France, April 22-26, 2003. Los Alamitos, CA: IEEE Computer Society, 2003. |
| **[Huff 1996a]** | Huff, K. E. "Effect of Product Lines on Current Process Technology," 5-7. *Proceedings of the 10th International Software Process Workshop*. Dijon, France, June 17-19, 1996. Los Alamitos, CA: IEEE Computer Society, 1996. |
| **[Humphrey 1992a]** | Humphrey, W. & Feiler, P. *Software Process Development and Enactment: Concepts and Defini-tions* (CMU/SEI-92-TR-004, ADA258465). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992. |

| | |
|---|---|
| **[Humphrey 1995a]** | Humphrey, W. *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley, 1995. (See page 4 for a definition of software process and pages 441-459 for information on process definition.) |
| **[Humphrey 2000a]** | Humphrey, W. "Justifying a Process Improvement Proposal." *news@SEI interactive 3*, 1 (March 2000). |
| **[IEEE 1987a]** | Institute of Electrical and Electronics Engineers. *IEEE Guide to Software Configuration Management* (IEEE Std 1042-1987). New York, NY: Institute of Electrical and Electronics Engineers, 1987. |
| **[IEEE 1990a]** | Institute of Electrical and Electronic Engineers. *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990). New York, NY: Institute of Electrical and Electronics Engineers, 1990. |
| **[IEEE 1996a]** | Institute of Electrical and Electronics Engineers. *IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools* (IEEE Std 1348-1995). New York, NY: Institute of Electrical and Electronics Engineers, 1996. |
| **[IEEE 1998a]** | Institute of Electrical and Electronics Engineers. *Information Technology–Guideline for the Evaluation and Selection of CASE Tools* (IEEE Std 1462-1998). New York, NY: Institute of Electrical and Electronics Engineers, 1998. |
| **[IEEE 1998b]** | Institute of Electrical and Electronics Engineers. *IEEE Guide for Information Technology–Systems Definition–Concept of Operations (CONOPS) Document* (IEEE Std 1362-1998). New York, NY: Institute of Electrical and Electronics Engineers, 1998. |
| **[IEEE 2000a]** | Institute of Electrical and Electronics Engineers. *Recommended Practice for Architectural Description of Software-Intensive Systems* (IEEE Std 1471-2000). New York, NY: Institute of Electrical and Electronics Engineers, 2000. |
| **[IEEE 2005a]** | Institute of Electrical and Electronics Engineers. *IEEE Standard for Software Configuration Management Plans* (IEEE Std 828-2005). New York, NY: Institute of Electrical and Electronics Engineers, 2005. |
| **[ISO 1995a]** | International Organization for Standardization. *Information Technology—Guideline for the Evaluation and Selection of CASE Tools* [ISO/IEC 14102:1995(E)]. Geneva, Switzerland: International Organization for Standardization, 1995. |
| **[ISO 1995b]** | International Organization for Standardization & International Electrotechnical Commission. *Quality Management—Guidelines for Configuration Management* [ISO 10007:1995 (E)]. Geneva, Switzerland: International Organization for Standardization/ International Electrotechnical Commission, 1995. |
| **[ISO 2007a]** | International Organization for Standardization. *International Organization for Standardization Home Page*. http://www.iso.org/iso/en/ISOOnline.frontpage (2007). |
| **[Jackson 2000a]** | Jackson, M. *Problem Frames and Methods: Structuring and Analyzing Software Development Problems*. New York, NY: Addison-Wesley, 2000. |
| **[Jacobson 1997a]** | Jacobson, I.; Griss, M.; & Jonsson, P. *Software Reuse: Architecture, Process, and Organization for Business Success*. Reading, MA: Addison-Wesley Longman, 1997. |
| **[Jandourek 1996a]** | Jandourek, E. "A Model for Platform Development." *Hewlett-Packard Journal 47*, 4 (August 1996): 56-71. |

| | |
|---|---|
| **[Jansen 2004a]** | Jansen, Anton & Bosch, Jan. “Evaluation of Tool Support for Architectural Evolution,” 375-378. *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE ‘04)*. Linz, Austria, September 20-24, 2004. Los Alamitos, CA: IEEE Computer Society Press, 2004. |
| **[John 2006a]** | John, I.; Knodel, J.; Lehner, T.; & Muthig, D. “A Practical Guide to Product Line Scoping.” *Software Product Lines: Proceedings of the 10th International Software Product Line Conference (SPLC 2006)*. Baltimore, Maryland, August 21-24, 2006. Los Alamitos, CA: IEEE Computer Society, 2006. |
| **[Jones 1996a]** | Jones, L. & Northrop, L. “The Establishing Phase: Planning for Successful Improvement.” *Software Process: Improvement and Practice 2*, 1 (March 1996): 51-53. |
| **[Jones 1996b]** | Jones, L.; Kasunic, M.; & Ginn, M. “Managing Technology Change: Implementing the SEI’s IDEAL Model in a Less Than Ideal World,” Track 5 [CD-ROM]. *Proceedings of the Eighth Software Technology Conference*. Salt Lake City, UT, April 21-26, 1996. Hill AFB, UT: Software Technology Support Center, in cooperation with Utah State University, Continuing Education, 1996. |
| **[Jones 1999a]** | Jones, L. & Northrop, L. “Software Process Improvement Planning,” 1-24. *Proceedings of the European Software Engineering Process Group Conference*. Amsterdam, Netherlands, June 7-10, 1999. UK: European Software Process Improvement Foundation, 1999. |
| **[Jones 2002a]** | Jones, L. & Soule, A. *Software Process Improvement and Product Line Practice: CMMI and the Framework for Software Product Line Practice* (CMU/SEI-2002-TN-012, ADA403868). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. |
| **[Jones 2004a]** | Jones, Lawrence G. *Software Process Improvement and Product Line Practice: Building on Your Process Improvement Infrastructure* (CMU/SEI-2004-TN-044, ADA431119). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. |
| **[Jones 2005a]** | Jones, L. & Northrop, L. *Software Product Line Adoption in a CMMI Environment* (CMU/SEI-2005-TN-028, ADA441309). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. |
| **[Kang 1990a]** | Kang, K.; Cohen, S.; Hess, J.; Novak, W.; & Peterson, A. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-021, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. |
| **[Kang 1998a]** | Kang, K.; Kim, S.; Lee, J.; Shin, E.; & Huh, M. “FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures.” *Annals of Software Engineering 5*, 5 (September 1998): 143-168. |
| **[Kang 2002a]** | Kang, K.; Lee, J.; & Donohoe, P. “Feature-Oriented Product Line Engineering.” *IEEE Software 19*, 4 (July/August 2002): 58-65. |
| **[Karlsson 1995a]** | Karlsson, E., ed. *Software Reuse: A Holistic Approach*. Chichester, England: John Wiley & Sons, 1995. |
| **[Kasse 2002a]** | Kasse, T. *Action Focused Assessment for Software Process Improvement*. Norwood, MA: Artech House, 2002. |
| **[Kazman 1998a]** | Kazman, R.; Klein, M.; Barbacci, M.; Lipson, H.; Longstaff, T.; & Carrière, S. J. “The Architecture Tradeoff Analysis Method,” 68-78. *Proceedings of the ICECCS*. Monterey, CA, August 10-14, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998. |

| | |
|---|---|
| **[Kazman 1999a]** | Kazman, R.; Barbacci, M.; Klein, M.; Carrière, S. J.; & Woods, S. G. “Experience with Performing Architecture Tradeoff Analysis,” 54-63. *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. Los Angeles, CA, May 16-20, 1999. New York, NY: ACM, 1999. |
| **[Kazman 2000a]** | Kazman, R.; Klein, M.; & Clements, P. *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, ADA382629). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. |
| **[Kazman 2002a]** | Kazman, R.; O’Brien, L.; & Verhoef, C. *Architecture Reconstruction Guidelines, Third Edition* (CMU/SEI-2002-TR-034, ADA412306). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. |
| **[Kenwood 2001a]** | Kenwood, Carolyn A. *A Business Case Study of Open Source Software*. http://www.mitre.org/work/tech_papers/tech_papers_01/kenwood_software/index.html (2001). |
| **[Keuffel 1999a]** | Keuffel, W. “Planning for and Mitigating Risk.” *Software Development 7*, 9 (September 1999): S1-S5. |
| **[Kirkpatrick 1992a]** | Kirkpatrick, R. J.; Walker, J. A.; & Firth, R. “Software Development Risk Management: An SEI Appraisal,” 1-28. *Software Engineering Institute Technical Review ‘92* (CMU/SEI-92-REV). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992. |
| **[Klein 1999a]** | Klein, M.; Kazman, R.; Bass, L.; Carriere, J., Barbacci, M.; & Lipson, H. “Attribute-Based Architecture Styles,” 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*. San Antonio, TX, February 22-24, 1999. Boston, MA: Kluwer Academic Publishers, 1999. |
| **[Knauber 2000a]** | Knauber, P.; Muthig, D.; Schmid, K.; & Widen, T. “Applying Product Line Concepts in Small and Medium-Sized Companies.” *IEEE Software 17*, 5 (September/October 2000): 88-95. |
| **[Knodel 2005a]** | Knodel, J. & Muthig, D. “Analyzing the Produce Line Adequacy of Existing Components,” 37-45. *R2PL 2005-Proceedings of the First International Workshop on Reengineering Towards Product Lines* (CMU/SEI-2006-SR-002, ADA448167). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. |
| **[Kotonya 1996a]** | Kotonya, G. & Sommerville, I. “Requirements Engineering with Viewpoints,” 150-163. *Software Requirements Engineering, Second Edition*. Los Alamitos, CA: IEEE Computer Society Press, 1996. |
| **[Krasner 1988a]** | Krasner, G. E. & Pope, S. T. “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.” *Journal of Object-Oriented Programming 1*, 3 (August/September 1988): 26-49. |
| **[Kruchten 1998a]** | Kruchten, P. *The Rational Unified Process: An Introduction*. Reading, MA: Addison-Wesley, 1998. |
| **[Kruchten 2004a]** | Kruchten, Philippe. *The Rational Unified Process: An Introduction, Third Edition*. Boston, MA: Addison-Wesley, 2004. |
| **[Krueger 2001a]** | Krueger, C. “Easing the Transition to Software Mass Customization,” 282-293. *Proceedings of the 4th International Workshop on Software Product Family Engineering*. Bilbao, Spain, October 3-5, 2001. New York, NY: Springer, 2001. |
| **[Krueger 2002a]** | Krueger, Charles W. “Variation Management for Software Production Lines,” 37-48. *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*. San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002. |

| | |
|---|---|
| **[Krut 96]** | Krut, R. & Zalman, N. *Domain Analysis Workshop Report for the Automated Prompt and Response System Domain* (CMU/SEI-96-SR-001, ADA311456). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Kuusela 2000a]** | Kuusela, J. & Savolainen, J. "Requirements Engineering for Product Families," 61-69. *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. Limerick, Ireland, June 4-11, 2000. New York, NY: ACM, 2000. |
| **[Lee 2000a]** | Lee, K.; Kang, K.; Koh, E.; Chae, W.; Kim, B.; & Choi, B. "Domain-Oriented Engineering of Elevator Control Software," 3-22. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Leon 2005a]** | Leon, Alexis. *Software Configuration Management Handbook, Second Edition*. Norwood, MA: Artech House, Inc., 2005. |
| **[Lim 1998a]** | Lim, W. C. *Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components*. Upper Saddle River, NJ: Prentice-Hall PTR, 1998. |
| **[Lewis 2005a]** | Lewis, G.; Morris, E.; O'Brien, L.; Smith, D.; & Wrage, L. *SMART: The Service-Oriented Migration and Reuse Technique* (CMU/SEI-2005-TN-029, ADA441900). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. |
| **[Livesey 1997a]** | Livesey, D. & Guinane, T. *Developing Object-Oriented Software: An Experience-Based Approach*. Upper Saddle River, NJ: Prentice-Hall, 1997. |
| **[Loveland-Link 1999a]** | Loveland-Link, J.; Barbour, R.; Krum, A.; & Neitzel, A. *Rollout and Installation of Risk Management at the IMINT Directorate, National Reconnaissance Office* (CMU/SEI-99-TR-009, ADA375848). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Low 1999a]** | Low, G. & Leenanuraksa, V. "Software Quality and CASE Tools," 142-150. *Proceedings of the Ninth International Workshop on Software Technology and Engineering Practice (STEP '99)*. Pittsburgh, PA, August 30-September 2, 1999. Los Alamitos, CA: IEEE Computer Society Press, 1999. |
| **[Lundell 2004a]** | Lundell, Björn & Lings, Brian. "Changing Perceptions of CASE Technology." *The Journal of Systems and Software 72*, 2 (2004): 271-280. |
| **[Maccari 2000a]** | Maccari, Alessandro & Riva, Claudio. "Empirical Evaluation of CASE Tools Usage at Nokia." *Empirical Software Engineering 5*, 3 (November 2000): 287-299. |
| **[Maccari 2002a]** | Maccari, Alessandro; Riva, Claudio; & Maccari, Francesco. "On CASE Tool Usage at Nokia," 59-68. *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE '02)*. Edinburgh, UK, September 23-27, 2002. Washington, DC: IEEE Computer Society Press, 2002. |
| **[Marttiin 1998a]** | Marttiin, P. & Koskinen, M. Similarities And Differences of Method Engineering and Process Engineering Approaches. http://citeseer.ist.psu.edu/marttiin98similarities.html (1998). |
| **[McAndrews 1993a]** | McAndrews, D. R. *Establishing a Software Measurement Process* (CMU/SEI-93-TR-016, ADA267896). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993. |
| **[McDonald 1995a]** | McDonald, M. & Dunbar, I. *Market Segmentation: A Step-by-Step Approach to Creating Profitable Market Segments*. Basingstoke, England: Macmillan Business, 1995. |
| **[McFeeley 1996a]** | McFeeley, R. *IDEAL: A User's Guide for Software Process Improvement* (CMU/SEI-96-HB-001, ADA305472). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |

| | |
|---|---|
| **[McGregor 1999a]** | McGregor, J. D. “Validating Domain Models.” *Journal of Object-Oriented Programming 12*, 4 (July/August 1999): 12-17. |
| **[McGregor 2001a]** | McGregor, J. D. *Testing a Software Product Line* (CMU/SEI-2001-TR-022, ADA401736). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. |
| **[McGregor 2002a]** | McGregor, J. D. “Building Reusable Test Assets for a Product Line,” 345-6. *Proceedings of the 7th International ICSR Conference*. Austin, TX, April 15-19, 2002. Berlin, Germany: Springer, 2002. |
| **[McGregor 2005a]** | McGregor, J. D. Arcade Game Maker Pedagogical Product Line: Concept of Operations, Version 2.0. (2005). |
| **[McGregor 2005b]** | McGregor, J. *Preparing for Automated Derivation of Products in a Software Product Line* (CMU/SEI-2005-TR-017, ADA448223). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. |
| **[Meyer 1997a]** | Meyer, M. H. & Lehnerd, A. P. *The Power of Product Platforms: Building Value and Cost Leadership*. New York, NY: The Free Press, 1997. |
| **[Microsoft 2007a]** | Microsoft Corporation. *Microsoft Home Page*. http://www.microsoft.com/en/us/default.aspx (2007). |
| **[Mockus 1999a]** | Mockus, A. & Siy, H. “Measuring Domain Engineering Effects on Software Change Cost,” 304-311. *Proceedings of Metrics 99: Sixth International Symposium on Software Metrics*. Boca Raton, FL, November 4-6, 1999. New York, NY: IEEE Computer Society Press, 1999. |
| **[Moore 1991a]** | Moore, G. A. *Crossing the Chasm, Marketing and Selling Technology Products to Mainstream Customers*. New York, NY: Harper Business Publishing, 1991. |
| **[Morris 1993a]** | Morris, C. & Ferguson, C. “How Architecture Wins Technology Wars.” *Harvard Business Review 71*, 2 (March-April 1993): 86-96. |
| **[Moszkowski 1986a]** | Moszkowski, B. *Executing Temporal Logic Programs*. New York, NY: Cambridge University Press, 1986. |
| **[Muller 1988a]** | Muller, H. & Klashinsky, K. “Rigi-A System for Programming-in-the Large,” 80-86. *Proceedings of the 10th International Conference on Software Engineering (ICSE)*. Raffles City, Singapore, April 11-15, 1988. New York, NY: IEEE Computer Society Press, April 1988. |
| **[Muller 2000a]** | Muller, H.; Jahnke, J.; Smith, D.; Storey, M.; Tilley, S.; & Wong, K. “Reverse Engineering: A Roadmap,” 47-60. *The Future of Software Engineering. Proceedings of the 22nd International Conference on Software Engineering*. Limerick, Ireland, June 4-11, 2000. New York, NY: ACM, 2000. |
| **[Muller 2003a]** | Muller, H.; Weber, A.; & Wong, K. “Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE),” 30-35. *Third International Workshop on Adoption-Centric Software Engineering* (CMU/SEI-2003-SR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. |
| **[Musa 1999a]** | Musa, J. *Software Reliability Engineering*. New York, NY: McGraw-Hill, 1999. |
| **[Newell 1972a]** | Newell, A. & Simon, H. A. *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall, 1972. |
| **[Northrop 2004a]** | Northrop, Linda. *Software Product Line Adoption Roadmap* (CMU/SEI-2004-TR-022, ADA431117). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. |

| | |
|---|---|
| **[O’Brien 2001a]** | O’Brien, L. & Stoermer, C. “MAP-Mining Architectures for Product Line Evaluations,” 35-44. *Proceedings of the Third Working IEEE/IFIP Conference on Software Architecture (WICSA1)*. Amsterdam, Netherlands, August 28-31, 2001. Los Alamitos, CA: IEEE Computer Society, 2001. |
| **[O’Brien 2002a]** | O’Brien, L.; Stoermer, C.; & Verhoef, C. *Software Architecture Reconstruction: Practice Needs and Current Approaches* (CMU/SEI-2002-TR-024, ADA407795). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. |
| **[O’Brien 2003a]** | O’Brien, L. & Stoermer, C. *Architecture Reconstruction Case Study* (CMU/SEI-2003-TN-008, ADA413856). Pittsburgh, PA: Engineering Institute, Carnegie Mellon University, 2003. |
| **[O’Brien 2005a]** | O’Brien, L.; Bass, L.; & Merson, P. *Quality Attributes and Service-Oriented Architectures* (CMU/SEI-2005-TN-014, ADA441830). Pittsburgh, PA: Engineering Institute, Carnegie Mellon University, 2005. |
| **[Oberndorf 1998a]** | Oberndorf, P. *COTS and Open Systems*. SEI Monographs on the Use of Commercial Software in Government Systems. Pittsburgh, PA: Software Engineering, Carnegie Mellon University, 1998. |
| **[Ohlinger 2000a]** | Ohlinger, J. CCT Lessons Learned: What We Did, Why We Did It, and What We Would Do Differently. http://sunset.usc.edu/GSAW/GSAW2000/pdf/Ohlinger.pdf (2000). |
| **[OMG 1996a]** | Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0* (97-02-25). Needham, MA: Object Management Group, Inc., 1996. |
| **[OMG 2005a]** | Object Management Group. *Software Process Engineering Metamodel Version 1.1*. http://www.omg.org/technology/documents/formal/spem.htm (January 2005). |
| **[OMG 2007a]** | Object Management Group. *UML Resource Page*. http://www.uml.org (2007). |
| **[OSI 2007a]** | The Open Source Initiative. *Open Source Home Page*. http://www.opensource.org/ (2007). |
| **[Ossher 2000a]** | Ossher, Harold; Harrison, William; & Tarr, Perri. “Software Engineering Tools and Environments: A Roadmap,” 261-277. *Proceedings of the Conference on the Future of Software Engineering*. Limerick, Ireland, June 4-11, 2000. New York, NY: ACM, 2000. |
| **[Osterweil 1987a]** | Osterweil, L. “Software Processes Are Software Too,” 2-13. *Proceedings of the 9th International Conference on Software Engineering (ICSE)*. Monterey, CA, March 30-April 2, 1987. New York, NY: IEEE Computer Society Press, 1987. |
| **[Park 1996a]** | Park, R. E.; Goethert, W. B.; & Florac, W. *A. Goal-Driven Software Measurement—A Guidebook* (CMU/SEI-96-HB-002, ADA313946). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Parnas 1972a]** | Parnas, D. “Information Distribution Aspects of Design Methodology,” 339-344. *Proceedings of the 1971 IFIP Congress*. Ljubljana, Yugoslavia, August 23-28, 1971. Amsterdam, Netherlands: North-Holland Publishing Company, 1972. |
| **[Parnas 2001a]** | Parnas, D. & Weiss, D. “Active Design Reviews: Principles and Practices,” 132-136. Weiss, D. & Hoffman, D., eds. *Software Fundamentals: Collected Papers by David L. Parnas*. Reading, MA: Addison-Wesley, 2001. |

| | |
|---|---|
| **[Paulk 1993a]** | Paulk, M.; Weber, C. V.; Garcia, S. M; Chrissis, M.; & Bush, M. *Key Practices of the Capability Maturity Model, Version 1.1* (CMU/SEI-93-TR-025, ADA263432). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993. |
| **[Paulk 1995a]** | Paulk, M. C.; Weber, C. V.; Curtis, B.; & Chrissis, M. B. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison-Wesley, 1995. |
| **[Peterson 1991a]** | Peterson, A. S. & Cohen, S. G. *A Context Analysis of the Movement Control Domain for the Army Tactical Command and Control System (ATCCS)* (CMU/SEI-91-SR-003, ADA248117). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1991. |
| **[Porter 1980a]** | Porter, M. *Competitive Strategy*. New York, NY: The Free Press, 1980. |
| **[Poulin 1997a]** | Poulin, J. S. *Measuring Software Reuse: Principles, Practices, and Economic Models*. Reading, MA: Addison-Wesley, 1997. |
| **[Poulin 1997b]** | Poulin, J. S. "The Economics of Software Product Lines." *International Journal of Applied Software Technology 3*, 1 (March 1997): 20-34. |
| **[Powell 1996a]** | Powell, A.; Vickers, A.; Williams, E.; & Cooke, B. Ch. 11, "A Practical Strategy for the Evaluation of Software Tools," 165-185. *Method Engineering: Principles of Method Construction and Tool Support—Proceedings of the IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering*. Atlanta, GA, August 26-28, 1996. London, UK: Chapman & Hall, 1996. |
| **[Prahalad 1990a]** | Prahalad, C. & Hamel, G. "The Core Competency of the Corporation." *Harvard Business Review 68*, 3 (May-June 1990): 79-92. |
| **[Pressman 1998a]** | Pressman, R. *Software Engineering: A Practitioner's Approach, Fifth Edition*. New York, NY: McGraw-Hill Book Company, 1998. |
| **[Prieto-Diaz 1991a]** | Prieto-Diaz, R. & Arango, G. *Domain Analysis and Software Systems Modeling*. Los Alamitos, CA: IEEE Computer Society Press, 1991. |
| **[Pronk 2000a]** | Pronk, B. J. "An Interface-Based Platform Approach," 331-352. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Radice 1994a]** | Radice, R. & Garcia, S. "Tutorial 4: Upgraded Integrated Process/TQM Tools: An Integrated Approach to Software Process Improvement," *The Sixth Annual Software Technology Conference Papers* [CD-ROM]. Salt Lake City, Utah, April 11-14, 1994. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994. |
| **[Ramesh 1997a]** | Ramesh, B.; Stubbs, C.; Powers, T.; & Edwards, M. "Requirements Traceability: Theory and Practice." *Annals of Software Engineering 3*, 3 (September 1997): 397-415. |
| **[Reifer 1997a]** | Reifer, D. J. *Practical Software Reuse: Strategies for Introducing Reuse Concepts in Your Organization*. New York, NY: John Wiley & Sons, 1997. |
| **[Robertson 1998a]** | Robertson, D. & Ulrich, K. "Planning for Product Platforms." *Sloan Management Review 39*, 4 (Summer 1998): 19-31. |
| **[Rolland 1997a]** | Rolland, C. *A Primer For Method Engineering*. http://citeseer.ist.psu.edu/cache/papers/cs/29244/ ftp:zSzzSzsunsite.informatik.rwth-aachen.dezSzpubzSzCREWSzSzCREWS-97-06.pdf/ rolland97primer.pdf (1997). |

| | |
|---|---|
| **[Ryans 2000a]** | Ryans, A.; More, R.; Barclay, D.; & Deutscher, T. *Winning Market Leadership: Strategic Market Planning for Technology-Driven Businesses*. New York, NY: John Wiley & Sons, 2000. |
| **[Schmid 2000a]** | Schmid, Klaus. "Scoping Software Product Lines," 513-532. *Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Schmid 2001a]** | Schmid, Klaus; Thiel, Steffen; Bosch, Jan; Johnsson, Susanne; Jaring, Michel; Thomé, Bernhard; & Trosch, Siegfried. *Scoping*. http://www.esi.es/esaps/public-pdf/CWD124-08-06-01.pdf (June 8, 2001). |
| **[Schmid 2005a]** | Schmid, K.; Krennrich, K.; & Eisenbarth, M. *Requirements Management for Product Lines: A Prototype* (IESE-Report 061.05/E). http://publica.fraunhofer.de/eprints/N-31550.pdf (2005). |
| **[Schmid 2006a]** | Schmid, K.; Krennrich, K.; & Eisenbarth, M. "Requirements Management for Product Lines: Extending Professional Tools," 113-122. *Proceedings of the Tenth International Software Product Line Conference (SPLC 06)*. Baltimore, Maryland, August 21-34, 2006. Los Alamitos, CA: IEEE Computer Society, 2006. |
| **[Schmidt 2000a]** | Schmidt, Douglas; Stal, Michael; Rohnert, Hans; & Buschmann, Frank. *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. New York, NY: Wiley, September 2000. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471606952.html |
| **[Schmidt 2003a]** | Schmidt, M. J. *What's a Business Case? And Other Frequently Asked Questions*. http://www.solutionmatrix.com/downloads/Whats_a_Business_Case.pdf (2003). |
| **[Schnell 1996a]** | Schnell, K.; Zalman, N.; & Bhatt, A. *Transitioning Domain Analysis: An Industry Experience* (CMU/SEI-96-TR-009, ADA310918). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Schwaber 2005a]** | Schwaber, Carey with the assistance of Barnett, Liz; Friedlander, David; & Hogan, Lindsey. *The Expanding Purview of Software Configuration Management*. Cambridge, MA: Forrester Research, Inc. <br><br> http://www.forrester.com/Research/Document/Excerpt/0,7211,36337,00.html (2005). |
| **[Seacord 2001a]** | Seacord, R.; Comella-Dorda, S.; Lewis, G.; Place, P.; & Plakosh, D. *Legacy System Modernization Strategies* (CMU/SEI-2001-TR-025, ADA396051). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. |
| **[Seacord 2003a]** | Seacord R., Plakosh D., Lewis G. A. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Boston, MA: Addison-Wesley, 2003 (ISBN 0-321-11884-7). |
| **[SEI 1995a]** | Software Engineering Institute. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, MA: Addison-Wesley, 1995. |
| **[SEI 2000a]** | Software Engineering Institute. *Capability Maturity Model Integration, Version 1.1 CMMI for Systems Engineering and Software Engineering (CMMI-SE/SW, V1.1)* (CMU/SEI-2000-TR-018, ADA388775). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. |
| **[SEI 2006a]** | Software Engineering Institute. *CMMI® for Development, Version 1.2 CMMI-DEV, V1.2* (CMU/SEI-2006-TR-008, ADA455858). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2006. |

| | |
|---|---|
| **[SEI 2007a]** | Software Engineering Institute. *Attribute-Driven Design Method* (2007). |
| **[SEI 2007b]** | Software Engineering Institute. *Software Architecture for Software-Intensive Systems* (2007). |
| **[SEI 2007c]** | Software Engineering Institute. *COTS-Based Systems (CBS) Initiative* (2007). |
| **[SEI 2007d]** | Software Engineering Institute. *Product Line Analysis* (2007). |
| **[SEI 2007e]** | Software Engineering Institute. *Product Line Technical Probe* (2007). |
| **[SEI 2007f]** | Software Engineering Institute. *Reengineering* (2007). |
| **[SEI 2007g]** | Software Engineering Institute. *Quality Attribute Workshops* (2007). |
| **[SEI 2007h]** | Software Engineering Institute. *Economics of Software Product Lines* (2007). |
| **[SEI 2007i]** | Software Engineering Institute. *Software and Systems Process Improvement Networks (SPINs)* (2007). |
| **[Shaw 1996a]** | Shaw, M. & Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, NJ: Prentice-Hall, 1996. |
| **[Shaw 2000a]** | Shaw, J. *Control Channel Toolkit: Open Architecture-Based Product Line Development* http://sunset.usc.edu/GSAW/GSAW2000/pdf/shaw.pdf (2000). |
| **[Sheard 1997a]** | Sheard, S. “The Frameworks Quagmire.” *Crosstalk 10*, 9. http://www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1997/09/frameworks.asp (September 1997). |
| **[Shehata 2002a]** | Shehata, M.; Eberlein, A.; & Hoover, J. *Requirements Reuse and Feature Interaction Management*. http://www2.enel.ucalgary.ca/People/eberlein/publications/FI_ICSSEA2002.pdf (2002). |
| **[Six Sigma 2007a]** | iSixSigma. *Six Sigma Home Page*. http://www.isixsigma.com/ (2007). |
| **[Smith 1990a]** | Smith, C. *Performance Engineering of Software Systems*. Reading, MA: Addison-Wesley, 1990. |
| **[Smith 2001a]** | Smith, C. & Williams, L. *Software Performance Engineering for Object-Oriented Systems*. Reading, MA: Addison-Wesley, 2001. |
| **[Smith 2002a]** | Smith, D.; O’Brien, L.; & Bergey, J. “Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line,” 316-327. *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*. San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002. |
| **[Sneed 2001a]** | Sneed, H. M. “Recycling Software Components Extracted From Legacy Programs,” 43-51. *Proceedings of the 4th International Workshop on Principles of Software Evolution*. Vienna, Austria, September 10-11, 2001. New York, NY: ACM Press, 2001. |
| **[Sommerville 1997a]** | Sommerville, I. & Sawyer, P. *Requirements Engineering: A Good Practice Guide*. New York, NY: John Wiley & Sons, 1997. |

| | |
|---|---|
| **[Soni 1995a]** | Soni, D.; Nord, R. R.; & Hofmeister, C. “Software Architecture in Industrial Applications,” 196-207. *Proceedings of the 17th International Conference on Software Engineering (ICSE)*. Seattle, Washington, April 23-30, 1995. New York, NY: Association for Computing Machinery Press, 1995. |
| **[SPC 1993a]** | Software Productivity Consortium. *Reuse Adoption Guidebook* (SPC-92051-CMC, Version 02.00.05). Herndon, VA: Software Productivity Consortium, 1993. |
| **[SPC 1993b]** | Software Productivity Consortium. *Reuse-Driven Software Processes Guidebook* (SPC-92019-CMC, Version 02.00.03). Herndon, VA: Software Productivity Consortium, 1993. |
| **[SPLiT 2004a]** | International Workshop on Software Product Line Testing. *SPLiT Proceedings and Presentations*. http://www.biglever.com/split2004/presentations.html (2004). |
| **[SPLiT 2005a]** | Second International Workshop on Software Product Line Testing. *SPLiT Proceedings and Presentations*. http://www.biglever.com/split2005/presentations.html (2005). |
| **[STARS 1996a]** | Software Technology for Adaptable Reliable Systems (STARS). *Organization Domain Modeling (ODM) Guidebook Version 2.0* (STARS-VC-A025/001/00). Manassas, VA: Lockheed Martin Tactical Defense Systems, 1996. |
| **[Sun 2007a]** | Sun Microsystems, Inc. *Sun Microsystems Home Page*. http://www.sun.com/ (2007). |
| **[Svahnberg 2000a]** | Svahnberg, M. & Bosch, J. “Issues Concerning Variability in Software Product Lines,” 50-60. *Proceedings of the Third International Workshop on Software Architectures for Product Families*. Las Palmas de Gran Canaria, Spain, March 15-17, 2000. Berlin, Germany: Springer, 2000. |
| **[Szyperski 2002a]** | Szyperski, C. *Component Software: Beyond Object-Oriented Programming, 2nd ed.*. Boston, MA: Addison-Wesley, 2002. |
| **[Thiel 2000a]** | Thiel, S. & Peruzzi, F. “Starting a Product Line Approach for an Envisioned Market: Research and Experience in an Industrial Environment,” 495-512. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Tichy 1989a]** | Tichy, N. & Charan, R. “Speed, Simplicity, Self-Confidence: An Interview with Jack Welch.” *Harvard Business Review 67*, 5 (September-October 1989): 112-120. |
| **[Tilley 1997a]** | Tilley, S. R. *Discovering DISCOVER* (CMU/SEI-97-TR-012, ADA331014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. |
| **[Tilley 1998a]** | Tilley, S. R. *A Reverse-Engineering Environment Framework* (CMU/SEI-98-TR-005, ADA343688). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. |
| **[Toft 2000a]** | Toft, P.; Coleman, D.; & Ohta, J. “A Cooperative Model for Cross-Divisional Product Development for a Software Product Line,” 111-132. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Tolvanen 2005a]** | Tolvanen, Juha-Pekka & Kelly, Steven. “Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences,” 198-209. *Proceedings of the Ninth International Software Produce Lines Conference (SPLC 2005)*. Rennes, France, September 26-29, 2005. New York, NY: Springer, 2005. |

| | |
|---|---|
| **[Tompkins 2004a]** | Tompkins, J. *40 Risks to Establishing an Outsourcing Relationship*. http://www.tompkinsinc.com/operations/info/40_Outsourcing_Risks.pdf (2004). |
| **[Tracz 1995a]** | Tracz, W. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. New York, NY: Addison-Wesley, 1995. |
| **[Tracz 1988a]** | Tracz, W. "RMISE Workshop on Software Reuse Meeting Summary," 41-53. *Software Reuse: Emerging Technology*. Washington, DC: Computer Society Press, 1988. |
| **[TreeAge 1999a]** | TreeAge Software, Inc. DATA Interactive White Paper. http://server.treeage.com/objdocs/start/whitepaper.php3 (1999). |
| **[Ulrich 2002a]** | Ulrich, W. M. *Legacy Systems: Transformation Strategies*. Upper Saddle River, NJ: Prentice Hall, 2002 (ISBN 0-13-044927-X). |
| **[Van Zyl 2000a]** | Van Zyl, J. & Walker, A. J. "Strategic Product Development: A Strategic Approach to Taking Software Products to Market Successfully," 86-111. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Vollman 1994a]** | Vollman, T. "Standards Support for Software Tool Quality Assessment," 29-39. *Proceedings of the Third Symposium on Assessment of Quality Software Development Tools*. Washington, DC, June 7-9, 1994. Los Alamitos, CA: IEEE Computer Society Press, 1994. |
| **[Vu 2000a]** | Vu, J. "Findings of the Managing Software Innovation and Technology Change Workshop: Managing Technology Transfer" [CD-ROM]. *Proceedings of the Software Engineering Process Group (SEPG) 2000*. Seattle, WA, March 20-23, 2000. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. |
| **[Wallnau 1997a]** | Wallnau, K.; Weiderman, N.; & Northrop, L. *Distributed Object Technology with CORBA and Java: Key Concepts and Implications* (CMU/SEI-97-TR-004, ADA327035). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. |
| **[Wallnau 2002a]** | Wallnau, Kurt; Hissam, Scott A.; & Seacord, Robert C. *Building Systems from Commercial Components*. Upper Saddle River, NJ: Addison-Wesley, 2002. |
| **[Walrad 2002a]** | Walrad, Chuck & Strom, Darrel. "The Importance of Branching Models in SCM." *IEEE Computer 35*, 9 (September 2002): 31-38. |
| **[Wappler 2000a]** | Wappler, T. "Remember the Basics: Key Success Factors for Launching and Institutionalizing a Software Product Line," 73-84. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Wartik 1992a]** | Wartik, S. & Prieto-Diaz, R. "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches." *International Journal of Software Engineering and Knowledge Engineering 2*, 3 (September 1992): 403-431. |
| **[Weiderman 1997a]** | Weiderman, N.; Northrop, L.; Smith, D.; Tilley, S.; & Wallnau, K. *Implications of Distributed Object Technology for Reengineering* (CMU/SEI-97-TR-005, ADA326945). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. |
| **[Weiss 1999a]** | Weiss, D. M. & Lai, C. T. R. *Software Product-Line Engineering: A Family-Based Software Development Process*. Reading, MA: Addison-Wesley, 1999. |

| | |
|---|---|
| **[Westfechtel 2003a]** | Westfechtel, Bernhard & Conradi, Reidar. "Software Architecture and Software Configuration Management," 24-39. *Proceedings of the ICSE Workshops SCM 2001 and SCM 2003: Selected Papers*. van der Hoek, A. & Westfechtel, B., eds. Lecture Notes in Computer Science Volume 2649. Berlin, Germany: Springer-Verlag, 2003. |
| **[Wheeler 2005a]** | Wheeler, David A. *Why Open Source Software/Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!* http://www.dwheeler.com/oss_fs_why.html (2005). |
| **[Wijnstra 2000a]** | Wijnstra J. "Supporting Diversity with Component Frameworks as Architectural Elements," 50-59. *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. Limerick, Ireland, June 4-11, 2000. New York, NY: ACM, 2000. |
| **[Wikipedia 2007a]** | Wikipedia. *Agile Software Development*. http://en.wikipedia.org/wiki/Agile_software_development (2007). |
| **[Williams 1999a]** | Williams, R.; Pandelios, G.; & Behrens, S. *Software Risk Evaluation (SRE) Method Description (Version 2.0) & SRE Team Members Notebook (Version 2.0)* (CMU/SEI-99-TR-029, ADA001008). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. |
| **[Wingerd 2005a]** | Wingerd, Laura & Seiwald, Christopher. *High-Level Best Practices in Software Configuration Management*. http://www.perforce.com/perforce/bestpractices.html (2005). |
| **[Withey 1996a]** | Withey, J. *Investment Analysis of Software Assets for Product Lines* (CMU/SEI-96-TR-010, ADA315653). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. |
| **[Woods 1999a]** | Woods, S. G.; Carriere, S. J.; & Kazman, R. "A Semantic Foundation for Architectural Reengineering," 391-398. *Proceedings of the ICSM99*. Oxford, UK, August 30 - September 3, 1999. Oxford, UK: Oxford Press, 1999. |
| **[Yacoub 2000a]** | Yacoub, S.; Mili, A.; Kaveri, C.; & Dehlin, M. "A Hierarchy of COTS Certification Criteria," 397-412. *Software Product Lines: Proceedings of the First Software Product Line Conference (SPLC1)*. Denver, Colorado, August 28-31, 2000. Boston, MA: Kluwer Academic Publishers, 2000. |
| **[Zahran 1998a]** | Zahran, Sami. *Software Process Improvement : Practical Guidelines for Business Success*. Reading, MA: Addison-Wesley, 1998 (ISBN 0-201-17782-X). |
| **[Zhang 2005a]** | Zhang, Weishan & Jarzabek, Stan. "Reuse Without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices," 57-69. *Proceedings of Software Product Lines: Ninth International Conference*. Rennes, France, September 26-29, 2005. New York, NY: Springer, 2005. |
| **[Zhao 1999a]** | Zhao, J. "Bibliography on Software Architecture Analysis." *Software Engineering Notes 24*, 3 (May 1999): 61-62. |
| **[Zona 1999a]** | Zona Research, Inc. *Enterprise JavaBeans Technology, a Business Benefits Analysis*. (1999). http://java.sun.com/products/ejb/pdf/zona.pdf |
| **[Zubrow 2003a]** | Zubrow, Dave & Chastek, Gary. *Measures for Software Product Lines* (CMU/SEI-2003-TN-031). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. |

## Acknowledgments

The principal team of authors of Version 5.0 thank the many others who also directly contributed. The following individuals were on the author teams of earlier versions: Len Bass, Joe Batman, Lisa Brownsword, Grady Campbell, Matthew Fisher, Brian Gallagher, Dennis Smith, Albert Soule, Scott Tilley, Nelson Weiderman, Steven Woods, and Dave Zubrow. Pennie Walters was the primary editor, Bob Fantazier and Daniel Pipitone created the illustrations, and Robert Krut formatted this framework for the Web.

## Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

**Phone**: 412/268.5800 | 888.201.4479
**Web**: www.sei.cmu.edu | www.cert.org
**Email**: info@sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.