Technical Report TR02-011

Elemental Design Patterns:
A Link Between Architecture and Object Semantics

Jason McC. Smith and David Stotts

Dept of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

smithja@cs.unc.edu

March 25, 2002

Elemental Design Patterns - A Link Between Architecture and Object Semantics

Jason McC. Smith
University of North Carolina at Chapel Hill
Sitterson Hall CB #3175
Chapel Hill, NC 27599-3175

smitha@cs.unc.edu

David Stotts
University of North Carolina at Chapel Hill
Sitterson Hall CB #3175
Chapel Hill, NC 27599-3175

stotts@cs.unc.edu

ABSTRACT

Design patterns are an important concept in the field of software engineering, providing a language and application independent method for expressing and conveying lessons learned by experienced designers. There is a large gap, however, between the aesthetic and elegance of the patterns as intended and the reality of working with an ultimately mathematically expressible system such as code. In this paper we describe a step towards meaningful formal analysis of code within the language of patterns, and discuss potential uses. The major contributions include: a compendium of Elemental Design Patterns (EDPs), a layer of seemingly simplistic relationships between objects that, on closer inspection, provide a critical link between the world of formal analysis and the realm of pattern design and implementation without reducing the patterns to merely syntactic constructs; an extension to the ς-calculus, termed ρ-calculus, a formal notation for expressing relationships between the elements of object oriented languages, and its use in expressing the EDPs directly. We discuss their use in composition and decomposition of existing patterns, identification of pattern use in existing code to aid comprehension, support for refactoring of designs, integration with traditional code analysis techniques, and the education of students of software architecture.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*design languages, object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns*; F.4.1 [**Mathematical Logic**]: [lambda calculus and related systems]; D.2.11 [**Software Engineering**]: Software Architecture—*patterns*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement—*restructuring, reverse engineering, and reengineering*; D.3.1 [**Programming Languages**]: Formal Definition and Theory

General Terms

design, languages, measurement, theory

Keywords

design patterns, elemental design patterns, sigma calculus, rho calculus, pattern decomposition, pattern identification, refactoring, education

1. INTRODUCTION

The history of programming is an exercise in hierarchical abstractions. As programming techniques have progressed in the field, language designers have continued to push the envelope of producing explicit constructs for those conceptual lessons learned in the previous generation of languages, and software architects have continued to build ever more complex and powerful abstractions. At the same time that these abstractions have established methods for producing well designed systems, they have created problems for the pure theorist and the accomplished practitioner alike, resisting attempts at the formalizations that are necessary for many critical analyses.

One of the current successful abstractions in widespread use is the design pattern, an approach that builds upon the nature of object oriented languages to describe portions of systems that designers can learn from, modify, apply, and understand as a single conceptual item[14]. Design patterns are generally, if informally, defined as common solutions to common problems which are of significant complexity to require an explicit discussion of the scope of the problem and the proposed solution. Much of the popular literature on design patterns is dedicated to these larger, more complex patterns, providing the practitioner with increasingly powerful constructs with which to work.

There is a class of problems, however, which is even more common, yet design patterns have in general ignored. These problems are usually considered too obvious to provide a description for, because they are in every good programmer's toolkit. The solutions to this class of problems we term *Elemental Design Patterns* (EDPs), and are the base concepts on which the more complex design patterns are built. Since they comprise the constructs which are used repeatedly within the more common patterns to solve the same problems, such as abstraction of interface and delegation of implementation, they exhibit some interesting properties for partially bridging the gap between the source code of every-

day practice and the higher level abstractions of the larger patterns. The higher-level patterns are thus described in the language of elemental patterns, which fills an apparent missing link in the abstraction chain.

Design patterns also present an interesting set of problems for the theorist due to their dual nature[2], with both formally expressible and informally amorphous halves. The concepts contained in patterns are those that the professional community has deemed important and noteworthy, and they are ultimately expressed as source code that is reducible to a mathematically formal notation. The core concepts themselves, however, have to date evaded such formalization. We show here that such a formalization is possible, and in addition that it can meet certain critieria we deem essential.

We assert that such a formal solution should be implementation language independent, much as the design patterns are, if it is to truly capture universal concepts of programming methodology. We further assert that a formal denotation for pattern concepts should be a larger part of the formal semantics literature. Patterns are built on the theory and concepts of object-oriented programming, as surely as object-oriented approaches are built on procedural theory. A formal representation of patterns should reflect this, allowing for more detailed analysis of the code body if it is desired, or allowing an analysis at a very high level of abstraction.

Sigma (ς) calculus provides the basis for such a system[1]. It is an object-oriented analogue to lambda calculus, and allows for a rich formal description of source code in a language independent manner. It suffers from a general difficulty of use, however, and is refined for our purposes by an extension we term *reliance operators* (or *rho (ρ) calculus*), which encode various relationships between classes, objects, methods, and fields in a form required for efficient searching for simple structural constructs.

We show how the $\varsigma + \rho$ calculus can be used to express our elemental design patterns directly and precisely, which in turn are used to express the more common class of design patterns. In essence, we build a well defined and formal chain from a basic denotational semantics for object based languages in general to the language of design patterns, providing a clear path between them, with well formed transformations and the opportunity for various interesting analyses of patterns and their applications within systems.

By allowing the discovery and analysis of design patterns in source code in a language, tool, and coding-style-independent way which preserves the semantics of the original patterns under translation to the various implementation requirements, we provide a rich and powerful tool to help engineers understand new systems, and to foresee ramifications of proposed changes to them. By rooting this system firmly in the established denotational semantics of object oriented theory, we produce opportunities for fully formal analysis at many levels of detail.

2. RELATED WORK

The decomposition and analysis of patterns is an established idea, and the concept of creating a hierarchy of related patterns has been in the literature almost as long as patterns themselves[9, 21, 31, 37]. The few researchers who have attempted to provide a truly formal basis for patterns have most commonly done so from a desire to perform refactoring of existing code, while others have attempted the more pragmatic approach of identifying core components of existing patterns in use.

2.1 Refactoring Approaches

Refactoring[13] has been a frequent target of formalization techniques, with fairly good success to date[10, 24, 27]. The primary motivation is to facilitate tool support for, and validation of, transformation of code from one form to another while preserving behaviour. This is an important step in the maintenance and alteration of existing systems, and patterns are seen as the logical next abstraction upon which they should operate.

2.1.1 Fragments

Fragments, as developed by Florijm, Meijers, and van Winsen[12], provide a practical implementation of pattern analysis and coding support in the Smalltalk language, and demonstrate the power of application of these concepts. Their *fragments* are abstractions of a design elements, such as classes, patterns, methods, or code, and contain roles, or *slots*, which are filled by other fragments. In this way they are bound to each other to produce an architecture in much the same way that objects, classes, and such are in a working system, but the single definition of a fragment allows them to work with all components of the system in a singular way. This approach, while successful in assisting an engineer in working with a system, does have some limitations. Detection of existing patterns in the system was deemed unlikely, due to the fact that "many conceptual roles did not exist as distinct program elements, but were cluttered onto a few, more complex ones." This indicates that there may be a lower level of conceptual roles to address, below fragments.

2.1.2 LePuS

Eden's work on LePuS[11] is an excellent example of formalizing a language for pattern description, based on the fragments theory. LePuS lacks a tie to the more traditional formal denotational semantics of language theory, however, severely limiting its usefulness as a unifying system for code analysis. Pattern analysis and metrics are but one portion of the spectrum of tasks associated with system maintenance and design. An exquisitely elegant architecture may in fact be a poor choice for a system where performance or some other criteria is of paramount importance. System architects must have available *all* the relevant information for their system to be able to appropriately design and maintain it. Legacy systems, in particular, benefit from more procedural-based analysis, including cohesion and coupling metrics, slice analysis, and other data-centric approaches. One goal of this research is to enhance the tools for refactoring, not only of current object-oriented systems, but of older code that may just now be being converted to an OO approach. It is therefore critical to incorporate such analysis systems into any pattern analysis framework. To not do so is to further the chasm between the abstract concepts and
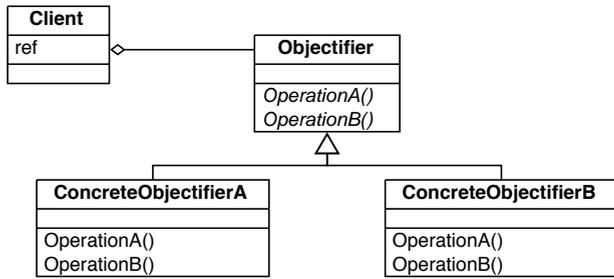
Figure 1: Objectifier class structure

Figure 2: Object Recursion class structure

the concrete code, instead of unifying them, as patterns are designed to do.

2.1.3 Minipatterns

Ó Cinnéide's work in transformation and refactoring of patterns in code[25] is an example of the application of *minipatterns*, portions of patterns that are used to compose larger bodies. Ó Cinnéide treats the minipatterns as stepping stones along a refactoring path, allowing each to be a discrete unit that can be refactored under a *minitransformation*, in much the same way that Fowler's refactorings[13] are used to incrementally transform code at and below the object level. These minipatterns are demonstrated to be highly useful for many applications, but cannot capture some of the more dynamic behaviour of patterns, instead relying heavily on syntactical constructs for evidence of the minipatterns.

2.2 Structural Analyses

An analysis of the 'Gang of Four' (GoF) patterns from the Design Pattens text [14] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor, for instance[14]. The relationships between patterns, such as inclusion or similarity, have been investigated by various practitioners, and a number of meaningful examples of underlying structures have been described. [4, 9, 31, 35, 36, 37]

2.2.1 Objectifier

The Objectifier pattern[37] is one such example of a core piece of structure and behaviour that is shared between many more complex patterns. Its Intent is to

> "Objectify similar behaviour in additional classes, so that clients can vary such behaviour independently from other behaviour, thus supporting variation-oriented design. Instances from those classes represent behaviour or properties, but not concrete objects from the real world (similar to reification)."

Zimmer uses the Objectifier as a 'basic pattern' in the construction of several other GoF patterns, such as Builder, Observer, Bridge, Strategy, State, Command and Iterator. It is a simple yet elegantly powerful structural concept that is used repeatedly in other patterns.
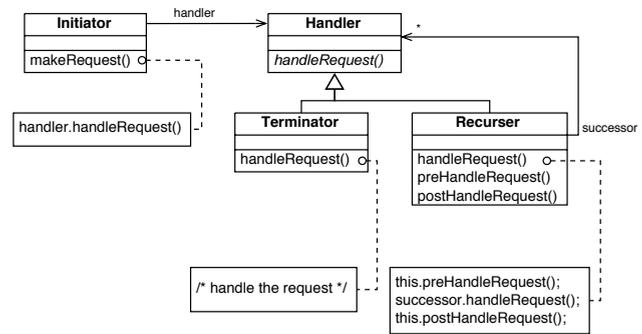
2.2.2 Object Recursion

Woolf takes this pattern one step further, adding a behavioural component, and naming it Object Recursion[36]. The class diagram in Figure 2 is extremely similar to Objectify, with an important difference, namely the behaviour in the leaf subclasses of *Handler*. Exclusive of this method behaviour, however, it looks to be an application of Objectify in a more specific use. Note that Woolf compares Object Recursion to the relevant GoF patterns and deduces that: Iterator, Composite and Decorator can, in many instances, be seen as containing an instance of Object Recursion; Chain of Responsibility and Interpreter do contain Object Recursion as a primary component.

2.2.3 Relationships

Taken together, the above instances of analyzed pattern findings seem to comprise two parts of a larger chain: Object Recursion contains an instance of Objectify, and both in turn are used by larger patterns.

This paper will answer the following questions: how far can we take this decomposition and recomposition of patterns in a meaningful way? Is it possible to continue to identify useful solutions to common problems *within* the patterns literature? Can these smaller pieces be considered true patterns, and why or why not? How finely can patterns be deconstructed? Is to do so useful, or merely a theoretical exercise?

3. EXAMINATION OF DESIGN PATTERNS

Our first task was to examine the existing canon of design pattern literature, and a natural place to start is the ubiquitous Gang of Four text[14]. Instead of a purely structural inspection, we chose to attempt to identify common concepts used in the patterns. A first cut of analysis resulted in eight identified probable core concepts:

AbstractInterface An extremely simple concept - you wish to enforce polymorphic behaviour by requiring all subclasses to implement a method. Equivalent to Woolf's Abstract Class pattern[35], but on the method level. Used in most patterns in the GoF group, with the exception of Singleton, Facade, and Memento.

DelegatedImplementation Another ubiquitous solution, moving the implementation of a method to another

object, possibly polymorphic. Used in most patterns, a method analog to the C++ *pimpl* idiom[9].

ExtendMethod A subclass overrides the superclass' implementation of a method, but then explicitly calls the superclass' implementation internally. It extends, not replaces, the parent's behaviour. Used in Decorator.

Retrieval Retrieves an expected particular type of object from a method call. Used in Singleton, Builder, Factory Method.

Iteration A runtime behaviour indicating repeated stepping through a data structure. May or may not be possible to create an appropriate pattern-expressed description, but it would be highly useful in such patterns as Iterator and Composite.

Invariance Encapsulate the concept that parts of a hierarchy or behaviour do **not** change. Used by Strategy and Template Method.

AggregateAlgorithm Demonstrate how to build a more complex algorithm out of parts that do change polymorphically. Used in Template Method.

CreateObject Encapsulates creation of an object, extremely similar to Ó Cinnéide's Encapsulate Construction minipattern[25]. Used in most Creational Patterns.

Of these, AbstractInterface, DelegatedImplementation and Retrieval could be considered simplistic, while Iteration and Invariance are, on the face of things, extremely difficult.

3.1 Method calls

On inspection, five of these possible patterns are centered around some form of method invocation. This led us to investigate what the critical forms of method calling truly are, and whether they could provide insights towards producing a comprehensive collection of EDPs. We assume, for the sake of this investigation, a dynamically bound language environment, and make no assumptions regarding features of implementation languages. Categorizing the various forms of method calls in the GoF patterns can be summarized as in Table 1, grouped according to four criteria:

Assume that an object a of type A has a method f that the program is currently executing. This method then internally calls another method, g, on some object, b, of type B. The columns represent, respectively, how a references b, the relationship between A and B, if any, the relationship between the types of f and g, whether or not g is an abstract method, and the patterns that this calling style is used in. Note that this is all typing information that is available at the time of method invocation, since we are only inspecting the types of the objects a and b and the methods f and g. Polymorphic behaviour may or may not take part, but we are not attempting a runtime analysis. This is strictly an analysis based on the point of view of the calling code.

If we eliminate the ownership attribute, we find that the table vastly simplifies, as well as reducing the information to strictly type information. In a dynamic language, the concept of ownership begins to break down, reducing the question of access by pointer or access by reference to a matter of implementation semantics in many cases. By reducing that conceptual baggage in this particular case, we are free to reintroduce such traits later.[1]

At this time, we can reorganize Table 1 slightly, removing the Mediator and Flyweight entry on the last line, as no typing attributable method invocations occur within those patterns. The result, shown in Table 2, is a list of eight method calling styles. Note that four of these are simply variations on whether the called method is abstract or not. By identifying this as an instance of the AbstractInterface component from above, we can simplify this list further to our final collection of the six primary method invocation styles in the GoF text, shown in Table 3. We will demonstrate later how to reincorporate AbstractInterface to rebuild the calling styles used in the original patterns.

A glance at the first column reveals that it can be split into two larger groups, those which call a method on the same object instance ($a = b$) and those which call a method on another object ($a \neq b$).

The method calls involved in the GoF patterns now can be classified by three orthogonal properties:

- The relationship of the target object instance to the calling object instance.

- The relationship of the target object's type to the calling object's type

- The relationship between the method signatures of the caller and callee

4. METHOD CALL EDPs

The first axis in the above list is simply a dichotomy between *Self* and *Other*.[2] The second describes the relationship between A and B, if any, and the third compares the types (consisting of a function mapping type, F and G, where $F = X \rightarrow Y$ for a method taking an object of type X and returning an object of type Y) of f and g, simply as another dichotomy of equivalence.

It is illustrative at this point to attempt creation of a comprehensive listing of the various permutations of these axes, and see where our identified invocation styles fall into place. For the possible relationships between A and B, we have started with our list items of 'Parent', where $A <: B$,[3] 'Sibling' where $A <: C$ and $B <: C$ for some type C, and 'Unrelated' as a collective bin for all other type relations at this

---

[1]Similarly, other method invocation attributes could be assigned, but do not fit within our typing framework for classification. For instance, the concept of constructing an object at some point in the pattern is used in the Creational Patterns: Prototype, Singleton, Factory Method, Abstract Factory, and Builder, as well as others such as Iterator and Flyweight. This reflects our CreateObject component, but we can place it aside for now to concentrate on the typing variations of method calls.

[2]*Child* is another possibility here, and a call to *Same* maps to BETA's *inner*, for example.

[3]The notation is taken from Abadi and Cardelli's sigma calculus[1]. $A <: B$ reads 'A is a subtype of B'

Ownership	Obj Type	Method Type	Abstract	Used In
N/A	self	diff	Y	Template Method, Factory Method
N/A	super	diff		Adapter (class)
N/A	super	same		Decorator
held	parent	same	Y	Decorator
held	parent	same		Composite, Interpreter, Chain of Responsibility
ptr	sibling	same		Proxy
ptr/held	none	none	Y	Builder, Abstract Factory, Strategy, Visitor
held	none	none	Y	State
held	none	none		Bridge
ptr	none	none		Adapter (object), Observer, Command, Memento
N/A				Mediator, Flyweight

Table 1: Method calling styles in Gang of Four patterns

Obj Type	Method Type	Abstract	Used In
self	diff	Y	Template Method, Factory Method
super	diff		Adapter (class)
super	same		Decorator
parent	same	Y	Decorator
parent	same		Composite, Interpreter, Chain of Responsibility
sibling	same		Proxy
none	none	Y	Builder, Abstract Factory, Strategy, Visitor, State
none	none		Adapter (object), Observer, Command, Memento, Bridge

Table 2: Reduced method calling styles in Gang of Four patterns

	Obj Type	Method Type	Used In
1	self	diff	Template Method, Factory Method
2	super	diff	Adapter (class)
3	super	same	Decorator
4	parent	same	Composite, Interpreter, Chain of Responsibility, Decorator
5	sibling	same	Proxy
6	none	none	Builder, Abstract Factory, Strategy, Visitor, State, Adapter (object), Observer, Command, Memento, Bridge

Table 3: Final method calling styles in Gang of Four patterns

point. To these we add 'Same', or $A = B$, as an obvious simple type relation between the objects.[4]

4.1 Initial list

We start by filling in the invocation styles from our final list from the GoF patterns, mapping them to our six categories in Table 3:

1. Self ($a = b$)

    (a) Self ($A = B$, or $a = this$)
        i. Same ($F = G$)..............................
        ii. Different ($F \neq G$)........Conglomeration[1]
    (b) Super ($A <: B$, or $a = super$)
        i. Same ($F = G$)............ExtendMethod[3]
        ii. Different ($F \neq G$).........RevertMethod[2]

2. Other ($a \neq b$)

    (a) Unrelated
        i. Same ($F = G$)....................Redirect[6]
        ii. Different ($F \neq G$)...............Delegate[6]
    (b) Same ($A = B$)
        i. Same ($F = G$)..............................
        ii. Different ($F \neq G$).........................
    (c) Parent ($A <: B$)
        i. Same ($F = G$)..........RedirectInFamily[4]
        ii. Different ($F \neq G$).........................
    (d) Sibling ($A <: C, B <: C, A \not<: B$)
        i. Same ($F = G$)..RedirectInLimitedFamily[5]
        ii. Different ($F \neq G$).........................

Each of these captures a concept as much as a syntax, as we originally intended. Each expresses a direct and explicit way to solve a common problem, providing a structural guide as well as a conceptual abstraction. In this way they fulfill the requirements of a pattern, as generally defined, and more importantly, given a broad enough context and minimalist constraints, fulfill Alexander's original definition as well as any decomposable pattern language can[2]. We will treat these as meeting the definition of design patterns, and present them as such.

The nomenclature we have selected is a reflection of the intended uses of the various constructs, but requires some defining:

Conglomeration Aggregating behaviour from methods of $Self$. Used to encapsulate complex behaviours into reusable portions within an object.

ExtendMethod A subclass wishes to extend the behaviour of a superclass' method instead of strictly replacing it.

---
[4] $Child$ is possible here as an addition as well, although we do not do so at this time.

RevertMethod A subclass wants *not* to use its own version of a method for some reason, such as namespace clash in the case of Adapter (class).

Redirect A method wishes to redirect some portion of its functionality to an extremely similar method in another object. We choose the term 'redirect' due to the usual use of such a call, such as in the Adapter (object) pattern.

Delegate A method simply delegates part of its behaviour to another method in another object.

RedirectInFamily Redirection to a similar method, but within one's own inheritance family, including the possibility of polymorphically messaging an object of one's own type.

RedirectInLimitedFamily A special case of the above, but limiting to a subset of the family tree, excluding possibly messaging an object of one's own type.

4.2 The full list

We can now begin to see where the remainder of the method call EDPs will take us. Again, we will present the listing, and briefly discuss each in turn.

1. Self ($a = b$)

    (a) Self ($a = this$)
        i. Same ($F = G$)...................Recursion
        ii. Different ($F \neq G$)..........Conglomeration
    (b) Super ($a = super$)
        i. Same ($F = G$)..............ExtendMethod
        ii. Different ($F \neq G$)...........RevertMethod

2. Other ($a \neq b$)

    (a) Unrelated
        i. Same ($F = G$).....................Redirect
        ii. Different ($F \neq G$)................Delegate
    (b) Same ($A = B$)
        i. Same ($F = G$)........RedirectedRecursion
        ii. Different ($F \neq G$)DelegatedConglomeration
    (c) Parent ($A <: B$)
        i. Same ($F = G$)............RedirectInFamily
        ii. Different ($F \neq G$)........DelegateInFamily
    (d) Sibling ($A <: C, B <: C, A \not<: B$)
        i. Same ($F = G$).....RedirectInLimitedFamily
        ii. Different ($F \neq G$).DelegateInLimitedFamily

Recursion Quite obvious on examination, this is a concrete link between primitive language features and our EDPs.

RedirectedRecursion A form of object level iteration.

DelegatedConglomeration Gathers behaviours from external instances of the current class.

DelegateInFamily Gathers related behaviours from the local class structure.

DelegateInLimitedFamily Limits the behaviours selected to a particular base definition.

5. OBJECT ELEMENT EDPs

At this point we have a fairly comprehensive array of method/object invocation relations, and can revisit our original list of concepts culled from the GoF patterns. Of the original eight, three are absorbed within our method invocations list: DelegatedImplementation, ExtendMethod, and AggregateAlgorithm. Of the remaining five, two are some of the more problematic EDPs to consider: Iteration, and Invariance. These can be considered sufficiently difficult concepts at this stage of the research that they are beyond the scope of this paper.

Our remaining three EDPs, CreateObject, AbstractInterface, and Retrieve, deal with object creation, method implementation, and object referencing, respectively. These are core concepts of what objects and classes are, and how they are defined. CreateObject creates instances of classes, AbstractInterface determines whether or not that instance contains an implementation of a method, and Retrieve is the mechanism by which external references to other objects are placed in data fields. These are the elemental creational patterns, and provide the construction of objects, methods, and fields. Since these are the three basic physical elements of object oriented programming[1], we feel that these are a complete base core of EDPs for this classification.[5]

CreateObject Constructs an object of a particular type.

AbstractInterface Indicates that a method has *not* been implemented by a class.

Retrieve Fetches objects from outside the current object, initiating external references.

The method invocation EDPs from the previous section are descriptions of how these object elements interact, defining the relationships between them. One further relationship is missing, however, that between types. Subtyping is a core relationship in OO languages, usually expressed through an inheritance relation between classes. However, subclassing is *not* equivalent to subtyping[1], and should be noted as a language construct extension to the core concepts of object-oriented theory. Because of this, we introduce a typing relation EDP, Inheritance, that creates a structural subtyping relationship between two classes. Not all languages directly support inheritance, it may be pointed out, instead relying on dynamic subtyping analysis to determine appropriate typing relations, such as in Emerald[17], or cloning mechanisms in prototype based languages such as Cecil[7] or NewtonScript[3].

Inheritance Enforces a structural relationship for subtyping.

6. THE EDP CATALOG

We believe our list of EDPs is now sufficiently comprehensive for useful analysis. We do not claim that this list covers all

---

[5]Classes, prototypes, traits, selectors and other aspects of various object oriented languages are expressible using only the three constructs identified.[1]

the possible permutations of interactions, but that these are the core catalog of EDPs upon which others will be built.

Object Element EDPs	
CreateObject	AbstractInterface
Retrieve	

Type Relation EDPs
Inheritance

Method Invocation EDPs	
Recursion	Conglomeration
Redirect	Delegate
ExtendMethod	RevertMethod
RedirectInFamily	DelegateInFamily
RedirectedRecursion	RedirectInLimitedFamily
DelegateInLimitedFamily	DelegatedConglomeration

6.1 Relationships are critical

At first glance, these EDPs seem highly unlikely to be very useful, as they appear to be positively primitive... and they are. These are the core primitives that underlie the construction of patterns in general. Patterns are, to be precise, descriptions of relationships between objects, according to Alexander[2], and method invocations and typing are the process through which objects interact. We believe that we have captured the elemental components of object oriented languages, and the salient relationships used in the vast majority of software engineering. If patterns are the frameworks on which to create large understandable systems, these are the nuts and bolts that comprise the frameworks.

And yet, each is unique from the others, each satisfies a different set of constraints, a different set of forces, and solves a slightly different problem. Each provides a degree of semantic context and a bit of conceptual elegance, in addition to a purely syntactical construct. In this context these are still truly patterns, and provide us with an interesting opportunity, to begin to build patterns from first principles of programming, namely formalizable denotation.

7. FORMALIZATION

Software historically has been rooted firmly in formal notations. Formal descriptions of software most decidedly lend themselves to a pattern's formal description using a formal notation. The entire pattern does not need to be given a formal form, nor would it be improved by doing so. The formal descriptions, however, should be as formal as possible without losing the generality that makes patterns useful. Source code is, at its root, a mathematical symbolic language with well formed reduction rules. We should strive to find an analogue for the formal side of patterns.

The question then arises as to how formal we can get with such an approach. A full, rigid formalization of static objects, methods, and fields would only be another form of source code, invariant under some transformation from the actual implementation. This defeats the purpose of patterns. We must find another aspect of patterns to encode as well, in order to preserve the flexibility of patterns.

7.1 Sigma Calculus

An analysis of desired traits for an intermediate formalization language includes that it be mathematically sound, consist of simple reduction rules, have enough expressive power to directly encode object-oriented concepts, and have the ability to flexibly encode relationships between code constructs.

Given these constraints, there are few options. The most obvious possible solution is lambda calculus or one of its variants [34], but lambda calculus cannot directly encode object-oriented constructs. Various extensions which would enable lambda calculus to do so have been proposed, but they invariably produce a highly cumbersome and complex rule set in an attempt to bypass apparently fundamental problems with expressing typed objects with a typed functional calculus [1]. One final candidate, sigma calculus, meets this requirement easily.

ς-calculus and its descendants are a fresh approach to creating a denotational semantics for object-oriented languages, and are "the first that does not require explicit reference to functions or procedures." [1] Defined and described in *A Theory of Objects* by Martín Abadi and Luca Cardelli, ς-calculus is an analogue to λ-calculus used in procedural theory. It concentrates on the aspects of object-oriented programming which are distinct from those of procedural programming, and makes no attempt to duplicate the efforts of the λ-calculus literature. Instead, it defines a notation providing a rigorous mathematical foundation for further object-oriented language theory. The prime elements are objects, methods and fields, the last two of which are treated as equivalent in ς-calculus, leading to some rather elegant solutions to some of the complex problems raised in formalizing highly dynamic languages. Classes are treated as a special case of objects, further simplifying the system.

ς-calculus is not an extension of λ-calculus. Attempts to produce such a hybrid have been made, but none has been particularly successful. A prime motivation for working to graft OO technologies onto λ-calculus is a desire to leverage off of the extremely large body of well done literature in that area. By starting anew, Abadi and Cardelli at first glance seem to have disposed of that body of work. On the contrary, they correctly recognize that the entirety of λ-calculus can be subsumed within the method calls of OOP. They even provide a mapping from λ-calculus to ς-calculus, resulting in "a simple and direct reduction semantics, instead of an indirect semantics involving both λ-abstraction and application." [1, p. 66]

7.1.1 *Inflexibility of ς-calculus*
While ς-calculus is a rich and important work in formalization of object oriented languages, it does not meet our needs for formalization of design patterns. As an example, we have translated the Singleton pattern, a very simple design pattern which is almost completely syntactical in form, to the ς-calculus, as shown in Figure 3.

As can be seen, this is a highly complex description for such a simple pattern. Not only is it extremely unwieldly, but it also suffers from a complete rigidity of form, and does not offer any room for interpretation of the implementation description, or any necessary fungibility that may be required for a specific application. This lack of adaptiveness means that there would be an explosion of definitions for just the Singleton pattern, each of which conformed to a single particular implementation. This breaks the distinction that patterns are implementation independent descriptions, as well as creating an excessively large library of possible pattern forms to search for in source code.

ς-calculus is not a particularly easy system or notation to learn. It is highly complex in abstractions, if not in implementation. The notation is cumbersome, and not intuitive without much study. To make matters worse, the definitive text on the subject is inscrutable, ambiguous, and difficult to read. Consequently, there are few practitioners of the ς-calculus at this time.

In addition to the practical concerns, ς-calculus fails the fourth requirement we set forth for our intermediate language: relationship encoding. The ς-calculus suite does not directly support this requirement.

7.2 Rho Calculus
It is fortunate then, that ς-calculus is simple to extend. We propose a new set of rules and operators within ς-calculus to support directly relationships and reliances between objects, methods and fields.

These *reliance operators*, as we have termed them, (the word 'relationship' is already overloaded in the current literature, and only expresses part of what we are attempting to deliver) are direct, quantifiable expressions of whether one element, (an object, method, or field) in any way relies or depends on the existance of another for its own definition or execution, and to what extent it does so.

This approach provides more detail than the formal description provided by UML, for instance. The calculus comprised of ς-calculus and these reliance operators, or *rho calculus*, maps nicely to the concepts of IsA, HasA, HoldsA, UsesA, and so on that exist within UML, indicating that a simple mapping between the two should exist. Unlike UML, however, reliance operators encode entire paths of reliances in a concise notation. All the reliances and relationships in the UML graphing system are encoded within the element that is under scrutiny, reducing the need for extended, and generally recursive, analysis for each element when needed.

Common concepts such as IsA in UML are directly expressable in ς-calculus using constructs such as, for instance for IsA, the transitive subsumption operator $B <: A$, indicating a relationship between a superclass (A) and a subclass (B). Other relationships, however, such as HasA, HoldsA, and UsesA, have no simple analogues in the base ς-calculus.

It is precisely these conceptual relations that are required to encode design patterns in such a way that they can be efficiently searched for by examining a drastically lesser set of constructs than would be necessary with raw ς-calculus.

We would like to continue the general notation of ς-calculus, so we adopt the operator used for subsumption, $<:$, analogous to IsA, and provide a similar sign, $\ll$, that indicates a reliance relationship. If $A \ll B$, then A relies on B in some

$$singleton \;=\; \Big[new = \varsigma(z) \Big[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n} \Big],$$
$$l_{1..n} = \varsigma(z)\lambda(s)b_i^{\,i\in 1..n},$$
$$uniqueInstance = \varsigma(z)\lambda(s)nil,$$
$$getInstance = \varsigma(z)\lambda(s)$$
$$s.uniqueInstance \rightarrow s.uniqueInstance;$$
$$s.uniqueInstance \Leftarrow s.new \,]$$

$$Singleton \;\triangleq\; [\, uniqueInstance : Singleton,$$
$$getInstance : Singleton \,]$$

$$Class(Singleton)_{Ins,Sub} \;\triangleq\; \Big[new : \Big[l_i : B_i^{\,i\in 1..n} \Big],$$
$$uniqueInstance : Singleton \rightarrow Singleton,$$
$$getInstance : Singleton \rightarrow Singleton \,]$$

$$class\Big(Singleton, a_i^{\,i\in I}\Big)_{Ins} \;\triangleq\; [new = \varsigma(z : Class(Singleton)_{Ins,I})$$
$$\Big[l_i := \varsigma(s : Singleton)z.l_i(s)^{i\in I} \Big],$$
$$uniqueInstance = \varsigma(z)\lambda(s)nil,$$
$$getInstance = \varsigma(z)\lambda(s)$$
$$s.uniqueInstance \rightarrow s.uniqueInstance;$$
$$s.uniqueInstance \Leftarrow s.new \,]$$

Figure 3: Singleton as expressed in ς-calculus

manner. It may be the interface, the implementation, a data member access, or a particular method call of B which is relied on by A for proper definition and operation. Differentiating between these paths of reliance is a bit more challenging.

It is important to note that we have developed an extension to an existing formal notation for object theory, which in turn is firmly linked to procedural theory. In this manner we acquire the capabilities of a vast body of knowledge and analysis techniques. We are not setting patterns up as a language unto themselves, but rather are showing a definite chain of abstractions, from the lowest, most concrete levels of programming, to the highest concepts with which system designers work.

For the purposes of this paper, however, we need only two reliance operators: First, $\ll_m$, indicating a method invocation call reliance. Given the expression $a.f \ll_m b.g$, it indicates that within the body of method f in object a, a call is made to method g of object b. Secondly, $<:$, the traditional inheritance (or more properly subsumption of type,) showing a type reliance.

7.3 RedirectInFamily

Consider the class diagram for the structure of RedirectInFamily, in Figure 4. Taken literally, it specifies that a class wishes to invoke a similar method (where, again, similarity is evaluated based on the function types of the methods) to the one currently being executed, and it wishes to do so on an object of a its parent class' type. This sort of open-ended structural recursion is a part of many patterns.

If we take the Participants specification of RedirectInFamily,
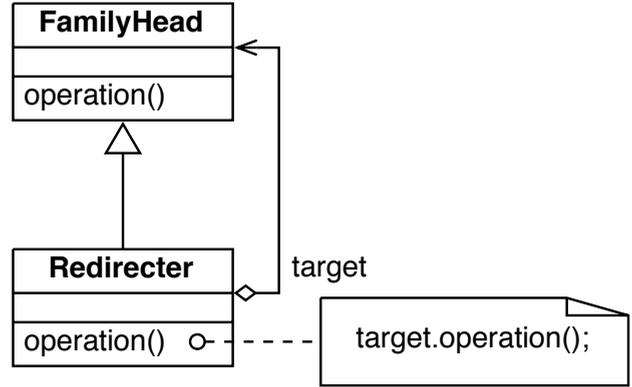

Figure 4: RedirectInFamily class structure

as described in the pattern in Appendix A, we find that:

- FamilyHead defines the interface, contains a method to be possibly overridden.

- Redirecter uses interface of FamilyHead through inheritance, redirects internal behaviour back to an instance of FamilyHead to gain polymorphic behaviour over an amorphous object structure.

We can express each of these requirements in ς-calculus:

$$FamilyHead \equiv [operation : A] \qquad (1)$$

$$Redirecter <: FamilyHead \qquad (2)$$

$$Redirecter \equiv [target : FamilyHead, \\ operation : A = \varsigma(x_i)\{target.operation\}] \qquad (3)$$

$$r : Redirecter \qquad (4)$$

$$fh : FamilyHead \qquad (5)$$

$$r.target = fh \qquad (6)$$

This is a concrete implementation of the RedirectInFamily structure, but fails to capture the reliance of Redirecter.operation on FamilyHead.operation's behaviour. So, we introduce our reliance operator $\ll_m$:

$$r.operation \ll_m r.target.operation \qquad (7)$$

We can reduce one level of indirection...

$$\frac{r.target = fh, r.operation \ll_m r.target.operation}{r.operation \ll_m fh.operation} \qquad (8)$$

...and now we can produce a necessary and sufficient set of clauses at this point to represent RedirectInFamily:

$$\frac{\begin{array}{c} Redirecter <: FamilyHead, \\ r : Redirecter, \\ fh : FamilyHead, \\ r.operation \ll_m fh.operation, \\ r.operation : A, \\ fh.operation : A \end{array}}{RedirectInFamily(r, fh, operation)} \qquad (9)$$

7.4 Isotopes

Common wisdom decrees that formalization of patterns in a mathematical notation will inevitably destroy the flexibility and elegance of patterns, reducing them to mere recipes and eliminating much of their usefulness. An interesting side effect of expressing our EDPs in the $\varsigma + \rho$-calculus, however, is an *increased* flexibility in expression of code while conforming to the core *concept* of a pattern.

Consider now Figure 5, where we have what, at first glance, doesn't look much like our original specification. We have introduced a new class to the system, our static criteria that the subclass' method invoke the method of the superclass' instance is gone, and a new calling chain has been put in place. In fact, this construction looks quite similar to the transitional state while applying Martin Fowler's *Move Method* refactoring[13].

We claim that this is precisely an example of a variation of RedirectInFamily, however, when viewed as a series of formal constructs, as follows, assuming the same class definitions given in equations 1 and 3:

$$Redirection <: FamilyHead \qquad (10)$$

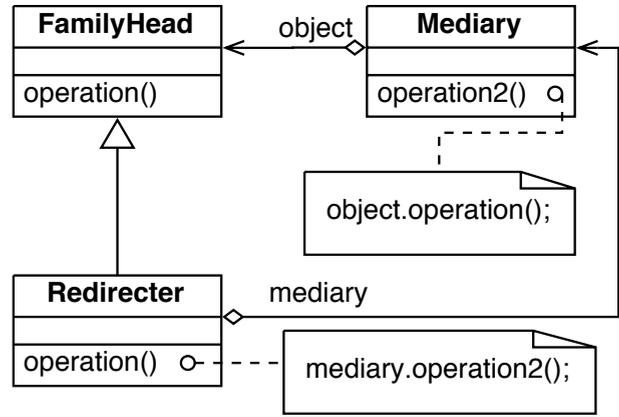**Figure 5: RedirectInFamily Isotope**

$$r : Redirection \qquad (11)$$

$$fh : FamilyHead \qquad (12)$$

$$m : Mediary \qquad (13)$$

$$r.mediary = m \qquad (14)$$

$$m.object = fh \qquad (15)$$

$$r.operation \ll_m r.mediary.operation2 \qquad (16)$$

$$mediary.operation2 \ll_m mediary.object.operation \qquad (17)$$

If we start reducing this equation set, we find that we can perform a transitive operation on Equations 16 and 17:

$$\frac{\begin{array}{c} r.operation \ll_m r.mediary.operation2, \\ mediary.operation2 \ll_m mediary.object.operation \end{array}}{r.operation \ll_m r.mediary.object.operation} \qquad (18)$$

We can now reduce this chain by equality substitutions from Equations 14 and 15:

$$\frac{\begin{array}{c} r.operation \ll_m r.mediary.object.operation, \\ r.mediary = m \end{array}}{r.operation \ll_m m.object.operation} \qquad (19)$$

$$\frac{r.operation \ll_m m.object.operation, m.object = fh}{r.operation \ll_m fh.operation} \qquad (20)$$

If we now take Equations 1, 3, 10, 11, 12, and 20, we find that we have satisfied the clause requirements set in our definition of RedirectInFamily, as per Equation 9. This alternate structure is an example of the RedirectInFamily pattern, without adhering to a strict class structure. We term this situation, where a variation on the original elemental design
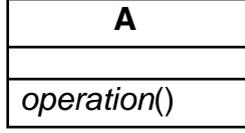
Figure 6: AbstractInterface

pattern class structure still conforms to the conceptual definition an *isotope*. The concepts of *object relationships and reliance* are the key.

An interesting side effect of this is that there is no explicit requirement that the relationships between *Redirecter* and *Mediary* or *Mediary* and *FamilyHead* be Redirection EDPs. In fact, they can be Delegation expressions, as we have shown above, with no change to the meaning of RedirectInFamily. Only the initial and terminus function signatures are important.

8. RECONSTRUCT KNOWN PATTERNS

We can now adequately demonstrate an example of using EDPs to express larger and well known design patterns. We begin with AbstractInterface, a simple EDP, and build our way up to Decorator, visiting two other established patterns along the way.

8.1 AbstractInterface

AbstractInterface is, simply put, ensuring that the method in a base class is truly abstract, forcing subclasses to override and provide their own implementations. The exceedingly simple class diagram for this is given in Figure 6. The ρ-calculus definition can be given by simply using the trait construct of ς-calculus:

$$\frac{A \equiv [new : [l_i : A \to B_i{}^i \in 1...n], operation : A \to B]}{AbstractInterface(A, operation)} \quad (21)$$

8.2 Objectifier

It should be obvious by now that Objectifier is simply a class structure applying the Inheritance EDP to an instance of AbstractInterface pattern, where the AbstractInterface applies to all methods in a class. This is equivalent to what Woolf calls an Abstract Class pattern. Referring back to Figure 1 from our earlier discussion in section 2.2.1, we can see that the core concept is to create a family of subclasses with a common abstract ancestor. We can express this in $\varsigma + \rho$-calculus as:

$$\frac{\begin{array}{c} Objectifier : [l_i : B_i{}^{i \in 1...n}], \\ AbstractInterface(Objectifier, l_i{}^{i \in 1...n}), \\ ConcreteObjectifier_j <: Objectifer^{j \in 1...m}, \\ Client : [obj : Objectifier] \end{array}}{Objectifier(Objectifier, ConcreteObjectifier_j{}^{j \in 1...m}, Client)} \quad (22)$$
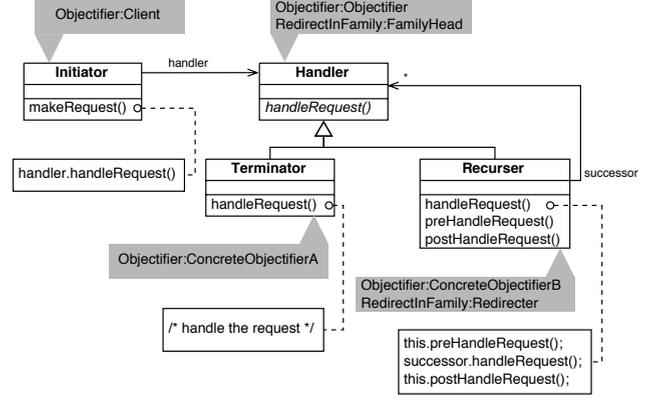

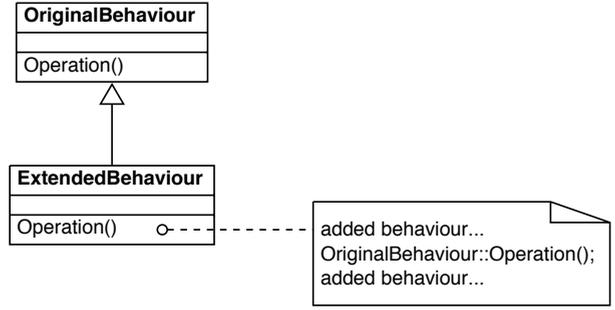Figure 7: Object Recursion, annotated to show roles

Figure 8: ExtendMethod class structure

8.3 Object Recursion

We briefly described Object Recursion in section 2.2.2, and gave its class structure in Figure 2. We now show that this is a melding of the Objectifier and RedirectInFamily patterns, as illustrated in Figure 7. The annotations indicate which roles of which patterns the various components of Object Recursion play. A formal EDP representation is:

$$\frac{\begin{array}{c} Objectifier(Handler, Recurser_i{}^{i \in 1...m}, Initiator), \\ Objectifier(Handler, Terminator_j{}^{j \in 1...n}, Initiator), \\ Initiator \ll_m obj.handleRequest, \\ obj : Handler, \\ RedirectInFamily(Recurser, Handler, handleRequest), \\ !RedirectInFamily(Terminator, Handler, handleRequest) \end{array}}{\begin{array}{c} ObjectRecursion(Handler, Recurser_i{}^{i \in 1...m}, \\ Terminator_j{}^{j \in 1...n}, Initiator) \end{array}} \quad (23)$$

8.4 ExtendMethod

The ExtendMethod EDP is used to extend, not replace, the functionality of an existing method in a superclass, as described in Section 4.1. Figure 8 shows the structure of such a pattern, illustrating the use of **super**. A formal definition can be given by:
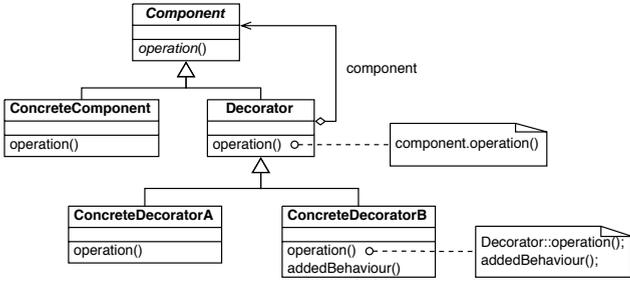
Figure 9: Decorator class structure

$$OriginalBehaviour : [l_i : B_i{}^{i \in 1...m}, operation : B_{m+1}],$$
$$ExtendedBehaviour <: OriginalBehaviour,$$
$$eb : ExtendedBehaviour,$$
$$eb.operation \ll_m \textbf{super}.operation$$

$$\overline{\quad ExtendMethod(OriginalBehaviour, \quad}$$
$$ExtendedBehaviour, operation)$$

$$(24)$$

8.5 Decorator

Now we can finally produce a pattern directly from the GoF text, the Decorator pattern. It is simple enough to be composed from the ground up, illustrating our technique of using fully formal methods entrenched in ς- and ρ-calculus coupled with the elemental design patterns catalog to create rich and conceptually true formal descriptions of useful design patterns. It is complex enough, however, to present a bit of a challenge, adding a bit of behavioural elegance to a primarily structural pattern.

Figure 9 is the standard class diagram for Decorator. Figure 10 shows the same diagram, but annotated to show how the ExtendMethod and Object Recursion patterns interact. Again, we provide a formal definition:

$$ObjectRecursion(Component, Decorator_i{}^{i \in 1...m},$$
$$ConcreteComponent_j{}^{j \in 1...n}, \textbf{any}),$$
$$ExtendMethod(Decorator, ConcreteDecoratorB_k{}^{k \in 1...o},$$
$$operation_k^{k \in 1...o}),$$
$$!ExtendMethod(Decorator, ConcreteDecoratorA_l{}^{l \in 1...p},$$
$$operation_l^{l \in 1...p})$$

$$\overline{\quad Decorator(Component, Decorator_i{}^{i \in 1...m}, \quad}$$
$$ConcreteComponent_j{}^{j \in 1...n},$$
$$ConcreteDecoratorB_k{}^{k \in 1...o},$$
$$ConcreteDecoratorA_l{}^{l \in 1...p},$$
$$operation_k^{k \in 1...o+p})$$

$$(25)$$

The keyword **any** indicates that any object of any class may take this role, as long as it conforms to the definition of Object Recursion.

Consider what we have just done - we have created a formally sound definition, from first principles, of a description of how to solve a problem of software architecture design. This definition is now subject to formal analysis, discovery,
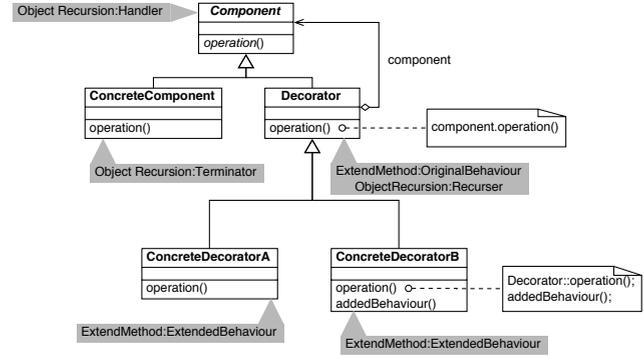

Figure 10: Decorator annotated to show EDP roles

and metrics, and, following our example of pattern composition, can be used as a building block for larger, even more intricate patterns that are *incrementally* comprehensible. At the same time, we believe we have retained the flexibility of implementation, as demonstrated with the RedirectInFamily isotope, that patterns demand. Also, we believe that we have retained the conceptual semantics of the pattern, by intelligently and diligently making precise choices at each stage of the composition. Furthermore, by building this approach on an existing denotational semantics for object oriented programming, ς-calculus, we continue to be able to process the same system at an extremely low level. Cohesion and coupling analysis[5, 16, 18, 19, 32], slice metrics production[20, 28, 29], and other traditional code analysis techniques[8, 10, 30] are still completely possible within the greater $\lambda + \varsigma + \rho$ calculus. We have provided the link between patterns, as conceptual entity descriptions, to the formal semantics required and used by compilers and other traditional tools, without losing the flexibility of implementation required by the patterns. We do not, however, see an explicit need to always resort to the full $\lambda + \varsigma + \rho$ calculus for all analysis. One of the key contributions of this system is that the practitioner can *choose* on which level to operate, and perform the analyses and tasks which are suitable without losing the flexibility of integrating other layers of analysis at a later date.

9. FUTURE DIRECTIONS

The future research possibilities now open to the software engineering community due to this work are many. They range across the full spectrum from formal analysis through human comprehension assistance, much as the $\varsigma + \rho$-calculus and elemental design patterns do.

9.1 EDPs as language design hints

It is expected that some will see the EDPs as truly primitive, but we would point out that the development of programming languages has been a reflection of directly supporting features, concepts, and idioms that practioners of the previous generation languages found to be useful. Cohesion and coupling analysis of procedural systems gave rise to many object oriented concepts, and each common OO language today has features that make concrete one or more EDPs. EDPs can therefore be seen as a path for incremental additions to future languages, providing a clue to which features programmers will find useful based precisely on what con-

cepts they currently use, but must construct from simpler forms.

9.1.1 Delegation

A recent, and highly touted, example of such a language construct is the *delegate* feature found in C#[23]. This is an explicit support for delegating calls directly as a language feature. It is in many ways equivalent to the decades old Smalltalk and Objective-C's selectors, but has a more definite syntax which restricts its functionality, but enhances ease of use. It is, as one would expect, an example of the Delegation EDP realized as a specific language construct, and demonstrates how the EDPs may help guide future language designers. Patterns are explicitly those solutions that have been found to be useful, common, and necessary in many cases, and are therefore a natural set of behaviours and structures for which languages to provide support.

9.1.2 ExtendMethod

Most languages have some support for this EDP, through the use of either static dispatch, as in C++, or an explicit keyword, such as Java and Smalltalk's **super**. Others, such as BETA[22], offer an alternative approach, deferring portions of their implementation to their children through the *inner* construct. Explicitly stating 'extension' as a characteristic of a method, as with Java's concept of *extends* for inheritance, however, seems to be absent. This could prove to be useful to the implementers of a future generation of code analysis tools and compilers.

9.1.3 AbstractInterface

The Abstract Interface EDP is, admittedly, one of the simplest in the collection. Every OO language supports this in some form, whether it is an explicit programmer created construct, such as C++'s pure virtual methods, or an implicit dynamic behaviour such as Smalltalk's exception throwing for an unimplemented method. It should be noted though that the above are either composite constructs (*virtual foo() = 0;* in C++) or a non construct runtime behaviour (Smalltalk), and as such are learned through interaction with the relationships between language features. In each of the cases, the functionality is not directly obvious in the language description, nor is it necessarily obvious to the student learning OO programming, and more importantly, OO design. Future languages may benefit from a more explicit construct.

9.2 Educational uses of EDPs

We believe the EDPs can provide a path for educators to guide students to learning OO design from first principles, demonstrating best practices for even the smallest of problems. Note that the core EDPs require only the concepts of classes, objects, methods (and method invocation), and data fields. Everything else is built off of these most basic OO constructs, which map directly to the core of UML class diagrams. The new student needs only to understand these extremely basic ideas to begin using the EDPs as a well formed approach to learning the larger and more complex design patterns. As an added benefit, the student will be exposed to concepts that may not be directly obvious in the language in which they are currently working. These

concepts are language independent, however, and should be transportable throughout the nascent engineer's career.

This transmission of best practices is one of the core motivations behind design patterns, but even the simplest of the usual canon requires some non-trivial amount of design understanding to be truly useful to the implementer. By reducing the scope of the design pattern being studied, one can reduce the background necessary by the reader, and therefore make the reduced pattern more accessible to a wider audience, increasing the distribution of the information. This parallels the suggestions put forth by Goldberg in 1994[15]. We are putting this into practice, incorporating the EDPs into the 2002-03 curriculum for software engineering at the University of North Carolina at Chapel Hill, to investigate the effectiveness of such an approach.

9.3 Semi-automated support for refactoring

Refactoring is not likely to ever be, in our opinion, a fully automatable process. At some point the human engineer must make decisions about the architecture in question, and guide the transformation of code from one design to another. Several key pieces, however, may benefit from the work outlined in this paper. Our isotope example in Section 7.4 indicates that it may be possible to support verification of Fowler's refactoring transforms through use of the combined $\lambda+\varsigma+\rho$-calculus, as well as various other approaches currently in use[13, 27, 24]. Ó Cinnéide's minitransformations likewise could be formally verified and applied to not only existing patterns, but perhaps to code that is not yet considered pattern ready, as key relationships are deduced from a formal analysis[25, 26]. Furthermore, we believe the fragments-based systems such as LePuS can now be integrated back into the larger domain of denotational semantics.

9.4 Comprehension of code

Finally, we revisit the original motivation for this research, to reduce the time and effort required for an engineer to comprehend a system's architecture well enough to guide the maintenance and modification thereof. We believe that the approach outlined in the paper, along with the full catalog of EDPs and rho calculus, can form a formal basis for some very powerful source code analysis tools such as *Choices*[33], or KT[6], that operate on a higher level of abstraction than just "class, object and method interactions"[33]. Discovery of patterns in an architecture should be become much more possible than it is today, and we expect that the discovery of *unintended* pattern uses should prove enlightening to engineers. In addition, the flexibility inherent in the $\varsigma + \rho$-calculus will provide some interesting possibilities for the identification of new variations of existing patterns.

10. CONCLUSION

We have presented here a suite of simple design patterns, the *elemental design patterns* and matching formalizations in the *ρ-calculus* for composition into larger, more useful and abstract design patterns as usually found in software architecture. These EDPs were identified initially through inspection of the existing literature on design patterns, establishing which solutions appeared repeatedly within the same contexts, mirroring the development of the more traditional design patterns. Their final state, however, is directly

analgous to the formal specifications of object oriented languages. We establish that these EDPs do conform to the definition of design patterns, and form the core building blocks to establish and describe the relationships between objects, methods and fields in object-oriented systems. Further, they are formally describable in the ρ-calculus, a notation that builds upon the ς-calculus, but adds the key concept of *reliance* to the base notation. These extensions, the *reliance operators* provide a large degree of flexibility to formally stating the relationships embodied in design patterns, without locking them into any one particular implementation.

These contributions will allow for new approaches to analyzing software systems, education of design patterns and best practices in object-oriented architecture, and may help guide future language design by indicating which design elements are most commonly used by software architects.

11. ACKNOWLEDGMENTS

12. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.

[2] C. W. Alexander. *Notes on the Synthesis of Form*. Oxford Univ Press, 1964. Fifteenth printing, 1999.

[3] Apple. *The NewtonScript programming language*. Apple Computer, Inc., 1993.

[4] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 1(2):18–52, May 1998.

[5] L. Briand and J. Daly. A unified framework for cohesion measurement in object-oriented systems. In *Proc. of the Fourth Conf. on METRICS'97*, pages 43–53, Nov. 1997. Albequerque.

[6] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, 2000.

[7] C. Chambers. The cecil language: Specification and rationale. Technical Report TR-93-03-05, University of Washington, 1993.

[8] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. cohesion/LCOM.

[9] J. Coplien. C++ idioms. In *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*, July 1998.

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 166–177. ACM Press, 2000.

[11] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in their Application*. PhD thesis, Tel Aviv University, Tel Aviv, Israel, 1999. Dissertation Draft.

[12] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In M. Askit and S. Matsuoka, editors, *Proc. of the 11th European Conf. on Object Oriented Programming - ECOOP'97*. Springer-Verlag, Berlin, 1997.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[15] A. Goldberg. What should we teach? In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, pages 30–37. ACM Press, 1995.

[16] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of ISACC'95*, pages 10–21, Insitut für Angewandte Informatik und Informationssysteme, Uni versity of Vienna, Rathaustraße 1914, A-1010 Vienna, Austria, 1995. Monterrey, Mexico.

[17] E. Jul, R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, and N. M. Hutchinson. Emerald: A general-purpose programming language. *Software Practice and Experience*, 21(1):91–118, Jan. 1991.

[18] B.-K. Kang and J. M. Bieman. Design-level cohesion measures: Derivation, comparison, and applications. In *Proc. 20th Intl. Computer Software and Applications Conf. (COMPSAC'96)*, pages 92–97, Aug. 1996.

[19] B.-K. Kang and J. M. Bieman. Using design cohesion to visualize, quantify and restructure software. In *Eighth Int'l Conf. Software Eng. and Knowledge Eng., SEKE '96*, June 1996.

[20] S. Karstu. An examination of the behavior of slice-based cohesion measures. Master's thesis, Minnesota Technological University, 2999.

[21] B. B. Kristensen. Complex associations: abstractions in object-oriented modeling. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 272–286. ACM Press, 1994.

[22] O. L. Madsen, B. Møller-Pederson, and K. Nygaard. *Object-oriented Programming in the BETA language*. Addison-Wesley, 1993.

[23] Microsoft Corporation, editor. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, 2002.

[24] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 235–250. ACM Press, 1996.

[25] M. Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach.* Ph.D. dissertation, University of Dublin, Trinity College, 2001.

[26] M. Ó Cinnéide and P. Nixon. Program restructuring to introduce design patterns. In *Proceedings of the Workshop on Experiences in Object-Oriented Re-Engineering, European Conference on Object-Oriented Programming, Brussels*, July 1998.

[27] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *Proc. of the Conf. on 1993 ACM Computer Science*, page 66, 1993. Feb 16-18, 1993, Indianapolis, IN.

[28] L. M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the 10th Annual Software Reliability Symposium, Denver, June 25-26, 1992*, June 1992.

[29] L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering, May 15-18, 1989*, May 1989.

[30] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Software Metrics Symposium, Baltimore, May 21-22 1993*, May 1993.

[31] D. Riehle. Composite design patterns. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 218–228. ACM Press, 1997.

[32] M. H. Samadzadeh and S. J. Khan. Stability, coupling and cohesion of object-oriented software systems. In *Proc. 22nd Ann. ACM Computer Science Conf. on Scaling Up*, pages 312–319, Mar. 1994. Mar 8-10, 1994, Phoenix, AZ.

[33] M. Sefika, A. Sane, and R. H. Campbell. Architecture-oriented visualization. In *Proceedings of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 389–405. ACM Press, 1996.

[34] R. Stansifer. *The Study of Programming Languages.* Prentice Hall, 1995.

[35] B. Woolf. The abstract class pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4 (Software Patterns Series).* Addison-Wesley, 1998.

[36] B. Woolf. The object recursion pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4 (Software Patterns Series).* Addison-Wesley, 1998.

[37] W. Zimmer. Relationships between design patterns. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.

APPENDIX

A. REDIRECTINFAMILY

RedirectInFamily Class Behavioral

Intent

Redirect some portion of a method's implementation to a possible cluster of classes, of which the current class is a member.
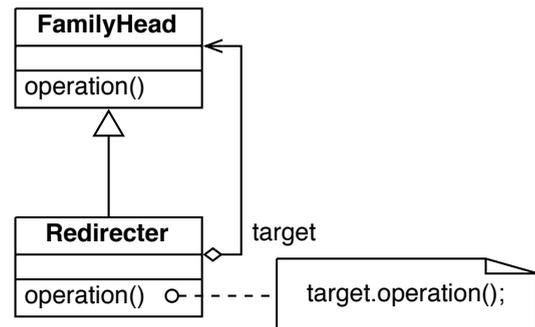
Motivation

Frequently a hierarchical object structure of related objects will be built at runtime, and behaviour needs to be distributed among levels.

Applicability

Use RedirectInFamily when:

- An aggregate structure of related objects is expected to be composed at compile or runtime.
- Behaviour should be decomposed to the various member objects.
- The structure of the aggregate objects is not known ahead of time.
- Polymorphic behaviour is expected, but not enforced.

Structure

Participants

FamilyHead
Defines interface, contains a method to be possibly overridden.

Redirecter
Uses interface of FamilyHead, redirects internal behaviour back to an instance of FamilyHead to gain polymorphic behaviour over an amorphous object structure.

Collaborations

Redirecter relies on the class FamilyHead for an interface, and an instance of same for an object recursive implementation.

Consequences

Redirecter is reliant on FamilyHead for portions of its functionality.

Implementation

In C++:

```
class FamilyHead {
public:
    virtual void operation();
};

class Redirecter : public FamilyHead {
public:
    void operation();
    FamilyHead* target;
};

void
Redirecter::operation() {
// preconditional behaviour
target->operation();
// postconditional behaviour
}
```

B.   EXTENDMETHOD

Extend Method   Object Behavioral

Intent

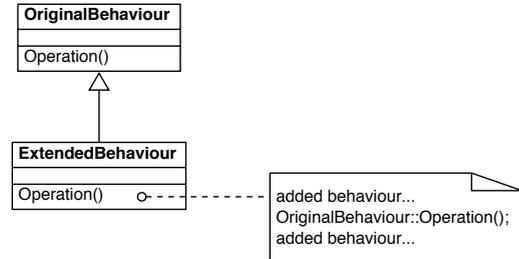Add to, not replace, behaviour in a method while reusing existing code.

Motivation

Often a behaviour should be augmented or extended in code. It is preferable to re-use code that preserves the original behaviour, instead of having to re-implement it. (Often the original code is unavailable, and we with to use existing methods as our base.)

Applicability

Use Extend Method when:

- Existing behaviour of a method needs to be extended but not replaced.
- Reuse of code is preferred or necessitated by lack of source code.
- Polymorphic behaviour is required.

Structure

Participants

OriginalBehaviour
Defines interface, contains a method with desired core functionality.

ExtendedBehaviour
Uses interface of OriginalBehaviour, re-implements method as call to base class code with added code and/or behaviour.

Collaborations

ExtendedBehaviour relies on OriginalBehaviour for both interface and core implementation.

Consequences

Code reuse is optimized, but the method Operation in OriginalBehaviour becomes fragile - its behaviour is now relied upon by ExtendedBehaviour::Operation to be invariant over time.

Behaviour is extended polymorphically and transparently to clients of OriginalBehaviour.

Implementation

In C++:

```
class OriginalBehaviour {
public:
    virtual void operation();
};

class ExtendedBehaviour : public OriginalBehaviour {
public:
    void operation();
};

void
OriginalBehaviour::operation() {
    // do core behaviour
}

void
ExtendedBehaviour::operation() {
```

```
        this->OriginalBehaviour::operation();
        // do extended behaviour
}
```

This pattern should translate very easily to most
any object-oriented language that supports inher-
itance and invocation of a superclass' version of a
method.