eXtreme Programming

An Overview

Methoden und Werkzeuge der Softwareproduktion WS 1999/2000

Author Thomas Dudziak

Figures

Pictures

Introduction

What is “eXtreme Programming” ?

eXtreme Programming (XP) is a software development process as well as a methodology.

A (software development) process defines who is doing what when and how. This means, it provides principles, techniques and practices for the efficient, predictable and repeatable production of software systems. Therefore, the process serves as a template for creating projects.
XP is also a process framework because it can be (and most likely will be) tailored to the specific needs of teams, projects, companies etc.

XP is also a lightweight methodology or what Alistair Cockburn calls a “Crystal Methodology”. In short, methodologies of this family have high productivity and high tolerance. Communication is usually strong with short paths, especially informal (not documented). There the is only a small range of deliverables (artifacts), but these are delivered frequently (releases). Processes of the Crystal family identify only a few roles and activities. For more information on the methodology space see [Coc].

To date, XP has been applied to business problems only, e.g. projects with a external customer that wants a specific product. The projects usually ranged from 6 to 15 months. XP was used by small teams ranging from two to twelve members (and it is likely to be limited to teams of this size).

Outline

The text is organized in three sections. In the first part the basic concepts of XP are presented. The second part discusses the process structure of XP. Finally, in the last section the practices and rules of the methodology eXtreme Programming are introduced.

Note that there isn’t much information available yet about the usage of XP. Although one introductionary book about XP has been published in 1999 ([Be99-1]) and several others are in the making, and the web sites give good introduction, especially information about the pros and cons of XP is still missing. Some possible problems are noted where applicable, however.

Basic Concepts

The four variables

XP regards a software development project as a system of four control “variables”: Cost, Time, Quality and Scope.
Note that these are only the names of the variables which XP identifies, not the general terms used Software Engineering.

- **Cost**
  The amount of money to be spent. The resources (how many developers, equipment etc.) available for the project are directly related to this variable.
- **Time**
  Determines when the system (release) should be done.
- **Quality**
  The correctness of the system (as defined by the customer) and how well tested it will be.
- **Scope**
  Describes what and how much will be done (functionality).

Time is the central variable in XP. The fundamental dependencies between it and the other variables are

- Increasing quality can increase the time that is needed because of more testing. Decreasing quality can reduce time to a certain degree (via reduction of the number of functional tests).
- Increasing cost (hiring more developers or providing better equipment) can mean less time but also the opposite effect is possible – for instance hiring more developers late in the project can increase time because of the overhead of communication.
  Decreasing cost increases time dramatically.
- Increasing scope means more time is needed because there is more work to do. Decreasing scope reduces time. This is the core control relationship in XP.

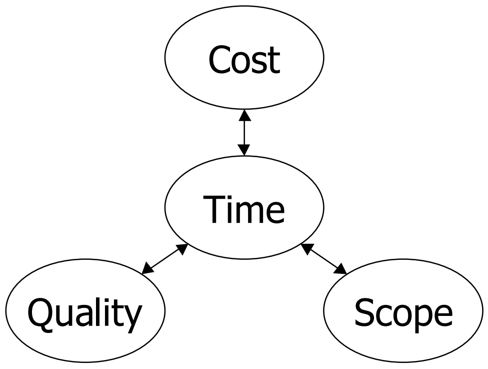


Figure 1 : The relationships between the variables

If for instance business, that is, management and customer, increases scope (more features) and holds onto cost (same resources) and quality, then time must be increased.
A special problem arises when scope is increased and time shall be fixed (schedule). Because you cannot control effectively with cost and quality, XP prevents this situation by explicitly stating that business can control cost, quality and either time or scope. The remaining variable is owned by development (which one remains depends on the Planning Game, p.8).

The four values

XP defines four “values” which are used as guidelines throughout development. These are

- **Communication**
  Good communication is one of the key factors necessary for the success of software projects. Customers need to communicate the requirements to the developers. Developers communicate ideas and design to each other and so on.

A lot of problems can be traced to a breakdown in communication (somebody forgot to ask the customer an important question, somebody forgot to communicate a change in the design etc.). This is not limited to direct communication, however. Documents (this does include the code) are an important way to communicate, as well.
XP tries to keep communication flowing in a variety of ways. Almost all practices rely on communication and emphasize it at the same time (see for instance Programming In Pairs, p.19 or Refactor Mercilessly, p.22).

- **Simplicity**
  XP strives for simple systems. This means, they should be as simple as possible but they must work.
  XP also strives for simplicity in the methodology. It reduces the amount of artifacts to an absolute minimum – the requirements (User Stories), plans (Planning Game) and the product (code). The practices and techniques can be learned in a matter of hours (although mastering them of course takes more time).
  The main reason for the desire for simplicity is that XP tries to cope with changes and other risks. Simplicity means that you always strive to "Do The Simplest Thing That Could Possibly Work" (p.21). XP is making the bet that it is better to do a simple thing today and pay a little more tomorrow to change it if necessary, than to do something more complicated today that may never be used later on anyway.
  Together, simplicity and communication work best. The simpler the system is, the easier it is to communicate it. The more you communicate the easier the system can get because you know more about the system.

- **Feedback**
  XP is a **feedback-driven** process. You need feedback at all scales, whether you are a customer, manager or developer. While coding you get immediate feedback from whitebox testing (Unit Tests). The customer defines blackbox tests (Functional Tests) and the team delivers releases frequently. From these practices, both the customers and the developers get feedback about the status of the system.
  Feedback has two important characteristics. The first is quality. Not only do you need to know that something is wrong but also what is wrong and what not. The second is time. The earlier and more often you get feedback the better. This way, problems are usually smaller and therefore corrections are cheaper.
  In XP feedback is especially relevant to business because it is the base for influencing the process.

- **Courage**
  This is a somewhat vague value. It includes courage as well as a certain amount of aggressiveness. Courage is needed because a lot of the rules and practices are "extreme" in the way that they go against "tradition" or "wisdom" of software engineering. XP also differs in the role of the customer in the process. He or she is much more involved in it (On-site Customer, p.20). For this to work, courage is required from both the customer and the developers.
  Aggressiveness is the attitude towards the implementation of the system. It drives for instance Refactor Mercilessly (changing the structure of the code whenever necessary).

These values can be seen in all rules and practices of XP. Take for instance Relentless Testing (p.20). Of course, the tests are a feedback mechanism. But they are also a means of communication, because most of them are code that give examples of the usage of units of the system. This way they provide insight into the design. Aggressiveness is also necessary for testing. The tests are (usually) implemented before the code they test and they try to cover as much as possible (and necessary).

Change-driven process

A **risk** is some "variable" that means a danger to the success of the project. Typical risks are for instance :

- Requirements change.
  This is one of the most problematic risks to software development. The assumption that the requirements are frozen is wrong in almost all cases. Some reasons for requirements changes are
  - The users needs change. This is especially true for long-term development.
  - The problem changes. Customers possibly don't know what they want, but they know what they do not want when they see it. This is known as the IKIWISI effect ("I'll know it when I see it").
  - The market changes. If the system is a competitive product it is possible that a (better) product which does the same work arrives during the development.
  - The requirements are imprecise or not fully understood. This is a danger especially for processes that strive to gather all requirements before the development phase (Waterfall model).

  In short, the real business value is learned, not known up-front. Therefore it is important to get feedback from customer often and early in order to deal with these changes.

- Technology
  New (and usually not well-known) or rapidly changing technology such as the internet present high risks for software projects. This is especially true for long-term development, although XP projects rarely exceed 15 months.
- Performance
  Systems can have requirements concerning for instance the response time or the work load.

XP explicitly deals with the first risk: “Embrace Change”. This means, the customer always has the right and the chance to change the requirements:

- The customer is part of the development team.
- He or she receives frequent releases so that he/she can see whether the system is what he wants
- He or she writes the requirements (User Stories)
- The team is honest. For every story it will confidently estimate how long it will take to implement the story so that the customer can decide how much is to be done or when it should be finished (see Planning Game).

The process of XP is designed to make a change of direction possible at any stage. However this means that projects for which most requirements can be determined in advance won’t profit from XP. These projects should probably be done with more conventional methodologies.

The Process XP

XP is an **iterative** and **incremental** process. The project is divided into smaller "mini-projects" which result in an increment of functionality, the so-called release. A **release** is a version of the planned system that makes business sense. All features that are part of the release are implemented completely.
An XP project creates **Frequent Releases** (every one to three months) in order to gain feedback early and often. Therefore the releases incrementally construct the desired functionality (the system grows over time).
Releases are negotiated in the Planning Games. Either the customer defines what should be part of the release and the developers determine how much time it will take to implement the release or the customer sets the schedule and the developers determine the amount of work that can be done in this time.

Each release cycle is constituted of a couple of iterations, each of which is at most three weeks long. The iteration is a primarily an organizational utility used to ease the necessary planning.

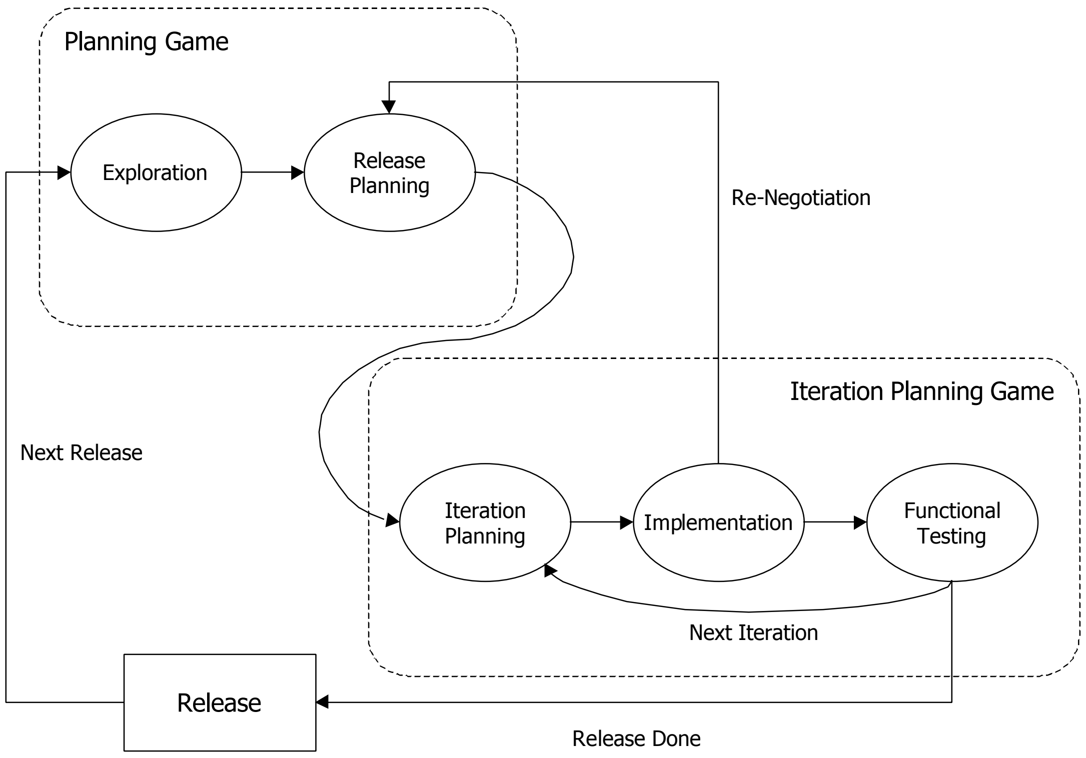


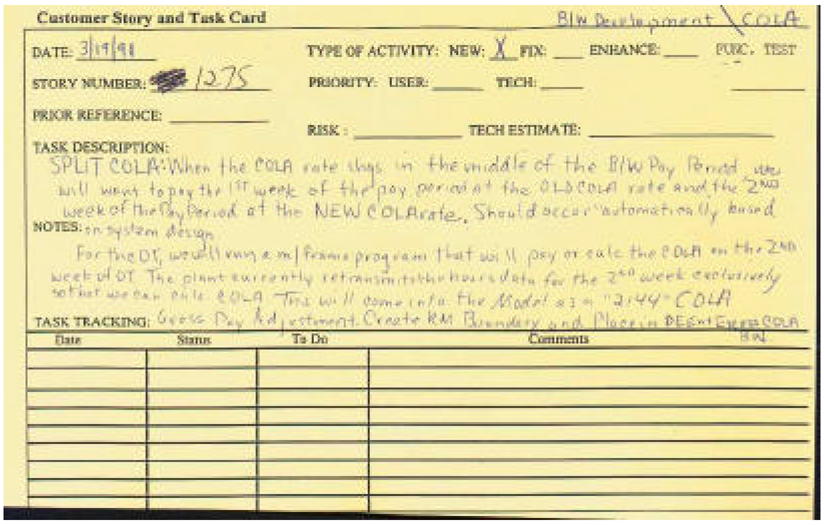
Figure 2 : Simplified process structure

Planning Game

The **Planning Game** consists of three steps : exploration, planning, and steering.
In the exploration phase the customer defines what he/she wants the system to do and the developers estimate how long it will take to implement the desired behavior.
In the planning phase both parties negotiate which of the desired features can be done within the given bounds (time, resources).
After the planning follows the steering phase. The negotiated plan is updated when necessary in response to what is learned in development and business while putting the plan in effect.

Exploration

The customer writes **User Stories** onto Index Cards. The stories will define what the customer wants the system to do, in other words they represent some functionality the user wants.


Customer Story and Task Card B/W Development \ COLA

DATE: 3/19/98 TYPE OF ACTIVITY: NEW: X FIX: ___ ENHANCE: ___ FUNC. TEST

STORY NUMBER: 1275 PRIORITY: USER: ___ TECH: ___

PRIOR REFERENCE: ___ RISK: ___ TECH ESTIMATE: ___

TASK DESCRIPTION:
SPLIT COLA: When the COLA rate chgs in the middle of the B/W Pay Period we will want to pay the 1st week of the pay period at the OLD COLA rate and the 2nd week of the Pay Period at the NEW COLA rate. Should occur automatically based

NOTES: on system design
For the DT, we will run a m/frame program that will pay or calc the COLA on the 2nd week of DT. The plant currently retransmits the hours data for the 2nd week exclusively so that we can calc COLA. This will come into the Model as a "2:44" COLA

TASK TRACKING: Gross Pay Adjustment. Create RM Boundary and Place in DEs in Excess COLA B/W

Date	Status	To Do	Comments

Picture 1 : A User Story card from the C3 project

They are comprised by a short name and one or two paragraphs of text. The text should avoid technical details (like database layouts etc.).
Each of the user stories will be estimated in terms of the time needed to implement it and the risk for the implementation (see Iteration, p.11). Therefore, the stories should be detailed enough to allow easy estimation.

For instance, assume the customer wants a security mechanism that provides access to the system only after a login with password. The user story for this login procedure should be something like :

> "When I first access the system it asks for a name and password. If they don't match, it shouldn't let me in. If they do match I should be able to use the system without entering the codes any more."

instead of

> " I want a login mechanism so we can prevent unauthorized access"

It is important that the development team helps the customer with at least the first couple of stories. Usually the customer is not accustomed to writing user stories so he or she needs feedback to learn what the developers need from the stories. The index cards help a lot here because they naturally "force" a specific amount of text to be written (an empty card does not look good, but the size of the card restricts the length of the text, as well).

Another desirable property of a user story is that it is testable. The reason for this is that Functional Tests are derived from the stories (see Iteration and Relentless Testing) which will help the customer verifying that the system does what it is expected to.

Typical projects of 6-12 months will create 50 to 100 stories. If you have less stories there will be a problem with setting priorities. More stories make the handling more difficult.

As said above, every story is estimated by the developers. The time estimation states how long the implementation of the story will take. It is given in so-called **Ideal Engineering Time** (IE Time). In short, one ideal engineering day is the amount of work that could be done by one developer when he or she is totally undisturbed and everything works smoothly.
Each story should result in 1 to 3 weeks of ideal engineering time. If the estimated time is longer, the story should be broken by the customer into smaller stories which are then re-estimated. If the estimated time is shorter the story will be combined with other short stories – but only in terms of estimation, not functionality (see Planning below).
One possible problem is that some user stories are not breakable in this way. It is not clear how XP deals with such stories.

If the developers do not understand the story good enough to estimate them confidently they will investigate it further and perhaps create one or more **Spike Solutions** to gain more knowledge. A spike solution is a small prototype that only inspects the problem at hand. This type of prototype is also used to experiment with other issues, for instance performance ("will the database be able to handle X requests per second"), or to choose between different strategies for a solution

Another benefit of exploration with Spike Solutions is that you get a better first guess for the Load Factor (see Planning below). Any spike solution created in this phase is estimated by the developers who implement them in terms of ideal engineering days it should take to complete the prototype. Then they measure how long it really took to complete them (in calendar days). From these data the initial guess for the load factor can be derived.

The whole exploration phase should take from a couple of weeks (when you have a team that knows each other as wells as the technology and domain) to a few months. A longer time span usually means that there are significant deficits in the understanding or communication which could probably be solved with a small, easy-to-complete pre-project which is done before the “real” project.

One argument against user stories is that they are not very detailed. In contrast to Use Cases, user stories are only a means of planning and gathering requirements. The details are filled in later on, when the story is about to be implemented and therefore is divided into Engineering Tasks.

Planning

When you have the set of stories from the customer along with the estimations from the developers, you’re ready to do the commitment schedule meeting. Here the **Commitment Schedule** is negotiated which is the set of stories that are to be implemented in the current release along with their estimates.

The customer makes three piles of stories :

- the stories necessary for the release to function
- the valuable stories (features that provide significant business value)
- the other stories (features that would be nice to have)

The Development tells Business the team velocity which is described by the **Load Factor**. This metric which is sometimes also called XPU (XP unit) describes the ratio between the real number of calendar days needed for task and the number of ideal engineering days estimated for the task. Initially the load factor is usually guessed to be 2 - 3 (calendar time divided by ideal engineering time) depending on how good the team knows each other and how much experience it has with XP and the business domain. When Spike Solutions were created during the exploration phase the load factor will be adjusted accordingly.
It is important to state that the load factor is not some goal to be reached but a metric which will change throughout the project - especially for the first release. Furthermore, Programming In Pairs has a profound impact on the load factor.

When both the piles of stories and the load factor are available, the commitment schedule is determined. The scope of the release is chosen by the customer in one of two possible ways :

- The release is to be done by a specific date.
  The developers will use the load factor and the estimation for the cards to determine the estimation in calendar time. For instance, a story has an estimation of 2 ideal engineering weeks and the load factor is at 2.5 which results in a calendar time estimation of 5 calendar weeks.
  When the estimations are ready, it is up to the customer to fill the available time with stories (which he or she would choose according to the piles).
- The customer wants a specific set of cards (usually the first two piles are completely in the set).
  As above, the developers estimate the stories chosen in calendar time and sum up the estimations to get the time needed to complete the set.

It is the right of the customer to change the scope (the set of stories) but not to change the load factor (which wouldn’t make sense anyway since it is a measured metric).
If both the customer and developers can commit to the set of cards and their estimations (therefore “commitment schedule”) the planning game meeting is finished.

Steering

The last phase of the planning game essentially makes up the rest of the cycle until the release is finished (or the project is cancelled).
Steering means influencing the process by little moves. The metaphor is derived from car-driving where you steer to stay on the road rather than point in the direction where the car should go.

It consists of four possible “moves” :

- **Iteration**
  The iterations make up the development.
- **Recovery**
  It is possible that development realizes that they cannot fulfill the commitment schedule (for instance the load factor is higher than estimated due to unexpected risks). It then has the right (and responsibility) to ask the customer for a re-negotiation of the commitment schedule which either results in a change of the release date or a change in scope (less stories).
- **New story**
  The customer has the right to add new stories. They will be estimated and – if the customer decides that they should be part of the current release – the commitment schedule will be re-negotiated.
- **Re-estimation**
  Development thinks that the plan is no longer accurate (to the good or the worse). Then all remaining stories are re-estimated and the commitment schedule is re-negotiated.

Development

After the plan is determined the release has to be implemented. This is done in the inner cycle.
First the commitment schedule is broken into iterations worth 2 – 3 calendar weeks (about one week ideal engineering time). The rule for breaking the schedule is that the most valuable stories are done first. It is worth noting that usually no conflicts arise from dependencies between the stories because the “foundation” stories tend to be more valuable to the business than the dependent stories.

Iteration

Each **Iteration** begins with an **Iteration Planning Game**. Similarly to the release planning game it consists of the phases exploration, planning, and steering.
Note that the planning for a particular iteration is performed at the beginning of the very iteration. No iterations are planned in advance. If something is important for a later iteration it is noted somewhere (onto a to-do list) so that it is at hand when the iteration is planned.

Usually every three iterations the Commitment Schedule is updated to reflect the status of the project. Especially important are new risks (e.g. a team member has left) and the refined Load Factor. It is possible that a new planning game is necessary in order to re-negotiate the commitment schedule.

On rare occasions a **Refactoring Iteration** is necessary (see Refactor Mercilessly). In such an iteration no new business value will be implemented but the system is re-structured to suit upcoming work. This is mostly due to unexpected risks or significant direction changes.

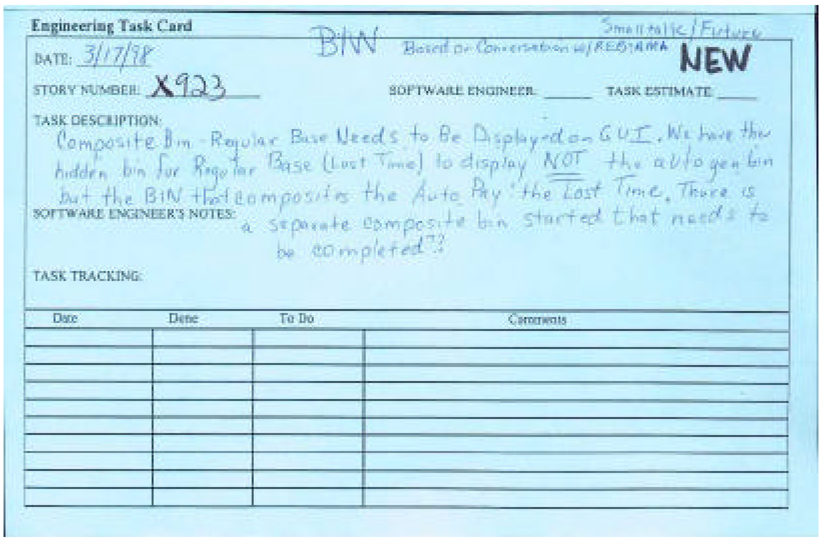
Exploration

From the outstanding part of the Commitment Schedule the customer chooses the stories to be done in this iteration. Usually these are the most valuable and probably most risky (“worst”) stories left. In addition, if there any stories whose functional tests failed in the last iteration are tackled, as well.
The developers break the stories into **Engineering Tasks** which are typically smaller than the stories. Sometimes a task results from more than one story or does not relate to a particular story at all (e.g. migrating to a new database version). It is a good practice to write the tasks onto index cards, similar to the stories.

Additional tasks come from the functional tests that failed in the last iteration.

For example consider the following user story (from [C3]) :

> “The RJ30 transaction records overtime in hours worked. The employee is to be paid time-and-a-half for overtime worked Monday through Saturday and double time for Sunday. Record time and dollars paid under regular, premium, and double premium. That is, a time-and-a-half hour puts one hour in regular, and one hour in premium, and same for the corresponding dollars. Sunday hours put one in regular and one in double premium.”


Engineering Task Card

Smalltalk / Future

BIN Based on Conversation w/ REB/AMA **NEW**

DATE: 3/17/98

STORY NUMBER: X923

SOFTWARE ENGINEER: ______ TASK ESTIMATE: ______

TASK DESCRIPTION:
Composite Bin - Regular Base Needs to Be Displayed on GUI. We have the hidden bin for Regular Base (Lost Time) to display NOT the auto gen bin but the BIN that composites the Auto Pay 'the Lost Time. There is

SOFTWARE ENGINEER'S NOTES:
a separate composite bin started that needs to be completed"?

TASK TRACKING:

Date	Done	To Do	Comments

Picture 2 : A Engineering Task card from the C3 project

The following tasks would result from it along with their estimates :

- Create input transaction definition for RJ30 record, placing record in HoursRawInput Bin. (0.5 day)
- Define new Bins for premium and double premium time and dollars. Add dollars Bins to gross pay composite Bin. (0.25 day)
- Create OvertimeEvaluationStation, determining for each overtime transaction whether the hours are regular, premium, or double premium. Put corresponding hours in appropriate bins. (0.5 day)
- Create OvertimePayStation, applying employee pay rate to hours, placing dollars into premium and double premium Bins as required. (Included in 0.5 day for OvertimeEvaluationStation.)

Note the use of the System Metaphor (p.24) – "Bin".

Planning

For each task a developer accepts responsibility. Tasks are not assigned but instead chosen voluntarily. Then the responsible developer estimates the task in Ideal Engineering Time. Only the developer that is responsible for the task performs the estimation.
A typical task should take between ½ and 3 IE days. If the task is shorter it is combined with other short tasks (only in terms of determining the schedule not the content of the task). If the task is longer it should be broken into shorter tasks if possible.
Note that the tasks are derived only from the stories to be done in this iteration. No coding in advance is performed.

The tasks combined with the name of the developers who signed for the tasks and their estimates form the **Iteration Schedule**. With the current Load Factor (as refined from previous iterations, see below) the team determines whether the iteration is over- or underbooked. If it is overbooked the customer has to shift stories from the current iteration to a later one. If it is under-booked the customer can select additional stories for this iteration.
Finally the work load is balanced. If a developer is overcommitted he has to give up some tasks.

Steering

In this phase the actual "coding" takes place. The following moves are part of it :

- **Implementation**
  A task card is implemented. See Rules And Practices for details
- **Record progress**
  XP defines a special role for measuring progress : the Tracker. See Roles and Metrics (p.14) for details.
- **Recovery**
  When it is found that a developer is overcommitted (either by the developer or the tracker), one or more of the following actions should be performed :

    - getting help from another partner (see Programming In Pairs) or doing a CRC Session to gain deeper understanding
    - reducing the amount of work (if possible by transferring tasks to other developers, otherwise by re-scheduling tasks for the next iteration)
    - asking the customer for reduction of the scope of the iteration (by moving stories to the next iteration). Note that this can mean re-negotiation of the iteration schedule.
- **Verification**
  Each story has associated functional tests. As soon as the tests are ready and all necessary tasks are implemented the tests are run to ensure that the story works (see Metrics).

At the end of an iteration all functional tests cases for the stories done in the iteration are reviewed with the customer. If any tests fail then the associated story will be part of the next iteration as well.

The Methodology XP - Rules and Practices

In contrast to the process the practices and rules used in XP for development are somewhat radical in comparison to more “conventional” methods.
All rules and practices work hand in hand. Although some of them could be (and are) practiced outside of XP (e.g. Unit Tests), the strength of XP lies in the combination of them. This is because the weaknesses of one practice or rule is covered by the strengths of other practices or rules.

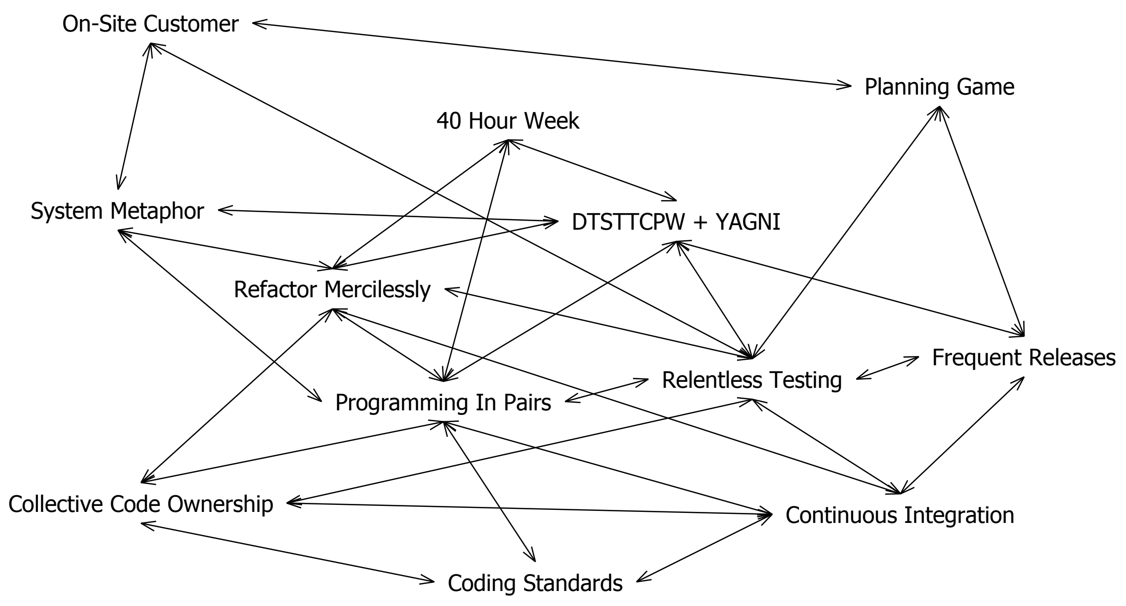


Figure 3 : Support relationships between the rules and practices

The rules and practices are ordered in four groups : management, development, design, and testing.

Management Rules And Practices

Metrics

XP defines two metrics which are used to describe the progress of the project.
The first metric is the **Load Factor**. As described in the Planning Game, it is the ratio between Ideal Engineering days and calendar days. Effectively it describes the team’s velocity.
The load factor is determined from the developer’s individual load factors which are repeatedly determined by the Tracker (see below). Good XP teams have a load factor about 2.5 meaning that they need 2.5 calendar days to do work worth one ideal engineering day. This may seem low but there are two things to take into account here:

- Programming In Pairs affects this number. Since most of the work is done in pairs this means that two people do the work one ideal engineering day worth. While programming in pairs you would usually work on the task of one partner in the first half of the day and in the next half on the task of another partner.
- IE time describes the amount of time that would be needed to accomplish the current task if it is the only thing to do and everything works out smoothly – all tests run first time, no disturbances etc.
  However, in reality there are other things to do as well, like integration, stand-up meetings etc.

It should be noted that the individual load factor of the coach is usually much higher (around 10 meaning that he or she needs 10 calendar days to produce work worth 1 ideal engineering day). This is due to the role (see below).

The other metric is the score of the Functional Test suite. While the load factor describes how “good” the team is working, this metric describes (within limits) the progress of the project.
For each story the customer writes one or more functional tests (with help from the developers). When all tasks for a specific story are done the functional tests for the story are run.

All these tests form a suite. The increase in the number of tests and the improvement of the scores describe the project velocity. Increasing scores give the customer the confidence that the system does what it is expected to do since he or she owns and defines the functional tests.
Usually you would create a graph of the scores and update it at determined times for instance at the end of every iteration.

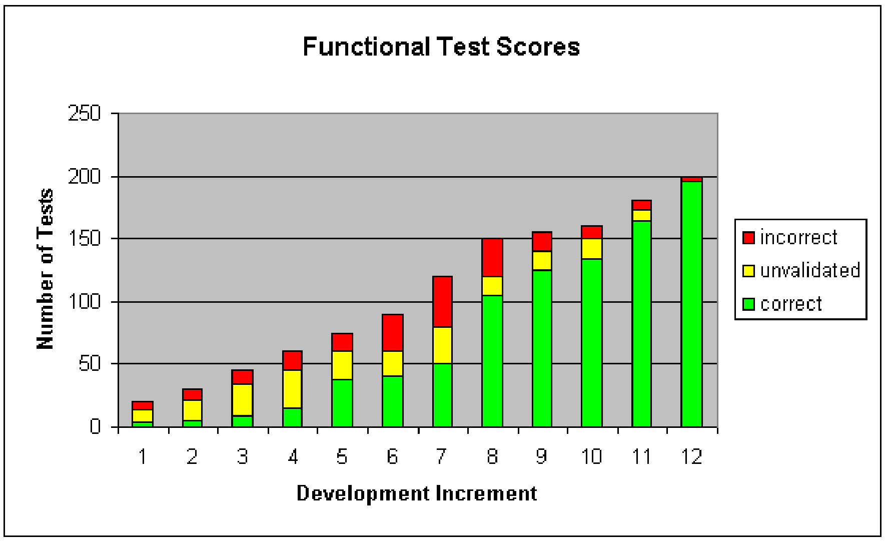


Figure 4 : Example of a Functional Test score graph

Roles

eXtreme Programming defines the following roles :

- **Customer**
  The customer defines what to do (User Stories) and in what order (Planning Games). However, in XP the customer is also responsible for the requirements because the stories are written by him or her. In addition, the Functional Tests are derived and verified by the customer from the stories (with the help from the Tester).
  XP has a special rule called On-Site Customer. This means, one representative of the customer, usually a domain expert who is a potential user of the system, is part of the development team and therefore at the development "site". The expert is then available for refining the user stories, derives functional tests etc.
  If the project has no real customer (for instance when creating off-the-shelf software) this role is played by the marketing department or an hired expert user.

- **Programmer**
  XP is a programmer-centric methodology. It does not make use of specialists like analysts, software architects or software designers. Instead this work is performed by the programmers.
  There are several skills that the programmer must have :
  - Communication
    As said before, XP relies on communication, especially face-to-face communication.
  - Coding
    Of course, the programmer should be reasonably good at coding. However, he or she does not have to be a "genius". In fact, master coders have more problems with XP than average programmers.
  - Ability to work in teams
    This is an obvious skill for a team-oriented methodology like XP. But it means more, especially with Collective Code Ownership and Programming In Pairs.

  Other skills can and will be learned in XP, such as Refactoring and Testing skills ("taste").

- **Coach**
  The managing part of an XP project is divided into two roles : the coach and the tracker.
  The coach is responsible for the technical execution (and evolution) of the project. He or she should have good communication skills and a "thick skin". The coach should also be technically skilled and confident. His or her job is to get everybody else to make good decisions, not to decide everything by himself or herself.
  The duties of the coach are :
  - explaining the process to management and customers

  - providing technical skills like testing, formatting (see Code Conventions), and Refactoring
  - having an overview of the system – particularly for long-term Refactoring goals
  - be available as a development partner (Programming In Pairs) especially for new team members

  Since the duties lie not so much in development, the load factor of the coach is typically much higher (around 10) than the individual load factors of the rest of the team (and would probably not be included in the overall load factor).
  The coach also participates in the management meetings for the project.

- **Tracker**
  The job of the tracker is to gather whatever metrics are being tracked for the project – at least the Load Factor and the Functional Test scores.
  The tracker will ask each developer every two or three days how much time they have spent on each of their tasks and how much time is left. This data is set in relation to the estimations for the tasks, and the load factor is updated. It is the tracker's responsibility to determine whether the Iteration Schedule and the Commitment Schedule can be met. The data is also used to give feedback to the developers about the quality of their estimates.
  In addition, the scores of the functional tests are gathered together with the Tester.
  Tracking is not really a full-time job. Therefore it is usually performed by the Coach or a Programmer.
  The tracker is sort of a mediator in the planning games so he or she must be versed in it. The tracker also participates in management meetings concerning the project.

- **Tester**
  In contrast to other processes, the tester has only few responsibilities. This is due to the fact that most of the whitebox testing (Unit Tests) is performed by the programmers. The tester helps the customer to choose and write the functional tests and run them (especially the tests that cannot be automated).
  Usually the role of the tester is not filled by a dedicated person but by one of the programmers or the tracker.

- **Consultant**
  Due to the rules and practices (especially Programming In Pairs), there are usually no specialists in XP projects. But sometimes the project needs deeper (technical) knowledge. In these cases a consultant will be hired who provides this knowledge. However, this role is different in XP. The consultant is hired to provide knowledge. One or two team members will sit with the consultant, ask a lot of questions about the technical domain and solve the problem. But the solution will be redone afterwards in order to share the knowledge among the team.

Workspace and Tools

XP teams are actually quite small (2-12 persons). For the teams one large room (**bullpen**) is used. All team members (programmers, coach, customer etc.) work together in this room.
The room has small cubbies at the sides and a set of computers in the middle. The cubbies are used to do CRC Sessions, talk with the on-site customer etc. One of the computers is used as a dedicated integration machine.

Continuous Integration needs a good Change Management Software which provide for easy integration and checkout of configurations ("snapshot").

Refactor Mercilessly is a lot easier and safer with tools that show possibilities for Refactoring and indicate whether a refactoring is safe or not. Perhaps the simple Refactorings (like renaming a method) can be performed by the tool. Such a **Refactoring Browser** is currently only available for Smalltalk.

Testing relies on automation. Only test suites that can be run easily (without the need for human control) and often (short execution time) will be run often. It is helpful to have a **Testing Framework** which provide the means for easy implementation of tests.

Picture 3 : The bullpen of the C3 project

Standup Meeting

Every day all team members (developers, customers etc.) meet at a specified time (C3's time is 10 AM) for a couple of minutes. Everybody briefly describes what he or she is working on, how it is going, interesting stuff that he or she found out, and any problems.

Picture 4 : A standup meeting

The standup meeting is an easy way to get help for problems, to announce interesting discoveries, to find partners for Programming In Pairs etc. However, it is not intended as a forum for solving the problems.

Forty hour Week

XP projects emphasize the forty hour week. This means that the team should avoid to work overtime.
Two reasons are given for this rule :

- "Projects that need working overtime will be too late anyway."
  Usually either the schedule is too tight (business controls both time and scope) or the team couldn't cope in time with unexpected risks. The only way to deal with the latter is to re-negotiate the schedule, otherwise the quality will be decreased.
- The motivation will be affected. For most of XP's rules and practices it is necessary that the team members are rested and motivated.

In practice, XP projects rarely require working overtime, and if so, at most one or two times during the whole project.

Development Rules And Practices

Development Cycle

Implementation occurs in small steps (tasks) for which XP defines the following procedure :

1. Analyze what is to be done. This probably involves analyzing Engineering Tasks and/or User Stories. If necessary a CRC Session is performed together.
2. Write Unit Tests. They will help in finding interfaces and they determine when the task is completed (Relentless Testing).
3. Implement just enough code to make the tests running. (Do The Simplest Thing That Could Possibly Work and You Aren't Gonna Need It)
4. Simplify the code if necessary. (Refactor Mercilessly)
5. Integrate the changes into the codebase. If problems occur resolve them (Continuous Integration). If the problems cannot be resolved, start over.

This whole procedure takes a couple of hours, at most a day.

Continuous Integration

Integration in XP refers to the activity in which changes to the system are combined with the **codebase** (the system currently stored in the change management). When some task is finished (adding a feature, fixing a bug, creating a unit test etc.) and the unit tests run 100 % the changes are integrated immediately. As said above the tasks usually take at most a day which means integration occurs every couple of hours. In fact, integration is part of the development cycle.
The benefits of frequent integration are :

- Integration is easier because the changes are small.
- The unit test suite ran fine before integration. If problems are discovered by the suite they probably are located in the changes.
- The codebase represents the current state of the system. If a snapshot is needed it can be created anytime from the codebase stored in the integration computer.

In "traditional" environments, code ownership and tools with check-in/check-out capabilities (probably with locking mechanisms) ensure that conflicts between developers editing the code are minimized.
In XP, the changes are integrated into the codebase whenever the developer thinks the task is finished, presumably every couple of hours, but at least once a day. Since the work is performed in small steps and integrated after every step, it is rather uncommon that there is a need to hold onto the changes for a longer time. However this can only work with Relentless Testing. Integration can only be done when the changes score 100% in the system's (unit) test suite. Collective Code Ownership is necessary as well. If there is a need to edit some code in order to make the changes or to integrate the developer must be able to do so.
Although one would think that a lot of editing conflicts arise, they rarely occur in practice. This is especially due to Refactor Mercilessly because Refactoring leads to a lot of small classes. Therefore the chance that two pairs are working on the same class are small.
If conflicts happen, the pair currently integrating is responsible for resolving them. Most of the time this is easy because the changes are small. If necessary one of the integrating pair can team up with one of the authors of the broken unit tests.
One possible problem is regression. A developer will only change a working piece of the system when he or she

- adds functionality
- refactors (some other part)
- fixes a bug (in some other part)

The test suites (unit and functional) are used to ensure that the functionality remains. Programming In Pairs can be used if necessary to resolve understanding conflicts.
To prevent conflicts arising from competing integration, usually one dedicated integration computer is used. With a decent change management software, every other developer can see whether he still works at the current version of a class or not (the term Configuration is used here to denote a snapshot of the system checked out for editing). If not the developers will have to check the configuration several times a day for changes.

Collective Code Ownership

The code (classes etc.) created in an XP project is owned by the complete team, not by the individual developers. This is important because anyone has to be able to modify anything if necessary. In other words, Collective Code Ownership is the foundation for the Development Cycle, especially for Refactor Mercilessly.
Of course, problems arise if Collective Code Ownership is not done in a disciplined way. In XP other practices provide for this discipline :

- Refactor Mercilessly increases the amount of units (classes, methods) in the system. This reduces the chance that more than one pair of developers work on the same unit at the same time.
- Continuous Integration is used to prevent conflicts or handle them if necessary. The integration must take place often to reduce the chance for conflicts.
- If a conflict occurs (for instance, two teams changed the same piece of code and one of the team has successfully integrated their changes which results in an integration problem for the second team), communication is used (probably via Programming In Pairs with one of the other team members) to resolve the conflicts.
- Coding Standards and Refactor Mercilessly ensure the readability of the system. This means changes are easier to make and understand.

However problems with Collective Code Ownership arise when

- a strong ownership mentality exists – this is reduced by the team “philosophy” of XP (Programming In Pairs etc.)
- non-project (in-house or legacy) components are (re)used – these components should not or cannot be subject of Collective Code Ownership
- expert knowledge is needed – Collective Code Ownership should be handled with care when the knowledge about a problem domain is not (yet) distributed among the team; Programming In Pairs and Relentless Testing is used to communicate the knowledge

Programming In Pairs

Any production code in an XP project is created by a pair of developers working together at one computer.
One is doing the actual coding (the so-called driver), e.g. creating classes, methods, unit tests etc.
The other is constantly reviewing the code (the “shotgun partner” or navigator). If the driver creates some code, with which the shotgun partner doesn’t agree, the driver has to justify it. In addition the navigator thinks “ahead”: about possible unit tests, or other solutions to problems etc. The navigator usually handles interruptions like questions from other developers, phone calls and the like.
Often the pair switch roles (“I have an idea for this – let me ride for a while.”). Furthermore, the pairs frequently change during the day, usually when some relatively independent task is finished (and integrated).

Picture 5 : A pair

Note that the pairs are not fixed at all. During the Iteration Planning Game every developer accepts responsibility for a couple of Engineering Tasks. When the developer starts working on the tasks needed for the Engineering Task, he or she will ask some other developer for help. The other developer could be more experienced in the domain of the problem at hand, or perhaps less experienced. Or the other developer simply can spare some time. If a developer is asked for help and can spare time he or she is obliged to help.
Both will pair up and implement one of the tasks. After a couple of hours they should be finished and they will switch pairs. The developer will probably be asked to help somebody else and therefore work on some other task than his own. In general developers will work on their own tasks half of the time and on tasks of other developers the other half.
There is nothing wrong with working alone, for instance to perform a web search or create a Spike Solution. But all work to be included in the code base has to come from a pair.

Pair Programming tend to create code of higher quality. Most of the little defects that creep into the code will be spotted by the shotgun partner. There is less indecision, and all decisions about the direction are made jointly. The other important gain is the distribution of knowledge. If everybody works with everybody else during the project, each time on probably different parts of the system (see Collective Code Ownership), he or she gets to know the system. This is also a great way to introduce new team members to the project. Furthermore, if the developer teams up with an (domain) expert, he or she can learn quite a bit about the domain, as well.

Programming In Pairs still presents a problem to managers. They ask why they should let two developers make the work of one if they would produce more working alone. However, recent studies (see [WiKe99], [WiKe] or [CoWi00]) have shown that pairs only need slightly more time (both partners summed up) for the same task than individual developers that worked alone – about 10% – but the code is of higher quality and there where less problems during implementation.
Nonetheless, this is still an area of research.

Coding Standards

XP projects use rules and guidelines for naming and formatting code units. This means, every developer chooses the names of classes, methods, variables etc. after these rules. The Coding Standards can come from the outside or are defined by the development team.
The standards make the system more consistent so that it is easier to read, understand, and work with (extend, refactor). It is also a great help when somebody (perhaps a new team member) needs to learn the system. Therefore it is important that every team member follows the rules. If an XP developer finds some part of the system that doesn't follow the standards, he or she changes it accordingly (this, of course, needs Collective Code Ownership).
Note that Coding Standards relate to the System Metaphor.

On-Site Customer

In almost all phases of XP communication with the customer is necessary. Therefore, a real customer who is familiar with the problem domain (if possible an expert) and who is a potential user of the system is part of the development team. He or she will sit with the team and be available to

- provide additional information necessary to implement a user story (story refinement)
- answer questions and resolve disputes about the meaning of domain aspects
- set priorities at a small scale (the choice of stories for an iteration)
- write functional tests and run them at the end of the iteration (with help from the Tester)

This gives the customer a simple method to steer the project.
Although business might say that they cannot afford a real customer at the team which could produce more value elsewhere, the on-site customer would produce value. The project will proceed faster and with less risk due to the immediate feedback. And no team can produce 40 hours of questions each week so that there will be time for normal work.

Relentless Testing

It is obvious that testing is one of the major building blocks of XP. Almost all practices rely on the safety net provided by frequent testing. The tests ensure that the system remains intact after changes and that it moves in the direction the customer wants it to.
XP has two levels of testing : Unit Testing which corresponds to whitebox or graybox testing, and Functional Testing which is blackbox testing. The [C3] system for example has over 3000 unit tests and over 600 functional tests.
Testing in XP is as much automated as possible. This means that the tests are implemented as code, if possible using test frameworks, and form an integral part of the codebase (and are therefore subject to other practices such as Continuous Integration and Refactor Mercilessly, although in a reduced fashion). The unit tests and the functional tests form one or two suites depending on whether all of the tests can be automated. A suite is always run completely. It should only provide information when some error was found.
The automation is encouraged because tests that can be executed without human interaction take less execution time and thus can be run more often. Furthermore, the tests communicate a lot about the usage of parts of the system.

Unit Tests are used to test “units” of code, e.g. methods and classes, that could possibly break in some way. The tests are run against expected or erroneous outcome to ensure the units are performing as expected. Simple units like accessors are usually not tested because the test would not bring much benefit.
Unit tests are written and run by a developer, usually the same one that will write or has written the unit to be tested. Unit testing is part of the Development Cycle as said above. The unit test suite is also changed or enhanced when a bug was found (to ensure that it never occurs again) or as the result of Refactorings.

Due to unit testing developers tend to program faster because the time spent in “bug hunting” and debugging is much shorter. Furthermore, writing unit tests in advance helps developing a stable and at the same time minimal interface of the unit. In combination with Refactor Mercilessly, the code is of higher quality, as well because you test as you develop and not later on.
In contrast to the other practices, unit testing can be performed independent of XP and the other practices (and provides the same benefits). However it gains from the XP practices – for instance from Collective Code Ownership (anybody can add unit tests when he or she feels that it is necessary) or Programming In Pairs (the testing tend to be more thorough).

Functional Tests are about verifying that the system in production meets the requirements of the customer (blackbox testing). For each story the customer writes one or more functional tests (with the help from development – Tester role) – this means that the customer is responsible for them (especially that they are correct). Their suite will be run at least at the end of each iteration.
The score is measured (see Metrics) and indicates the progress of the project. However the score does not have to be 100% (as for unit tests) – the customer decides whether the score is acceptable. If not, then the customer must decide which of the failures have to be tackled in the next iteration.

Design Rules And Practices

The most obvious difference between XP and other processes is that XP does no “up-front” design at all. Up-front means that before any implementation takes place the system is defined in terms of architecture and design models. XP in contrast relies on several practices, especially Refactoring and System Metaphor, to achieve a stable and simple system structure.
The communication of such not-well documented systems seems to be more difficult. However, it could be possible to retrieve design documents automatically from the system when needed (reverse engineering). In addition, the other practices (Programming In Pairs etc.) provide a much deeper knowledge of the system.

Do The Simplest Thing That Could Possibly Work (DTSTTCPW)

A developer should implement the simplest thing that he or she can think of and that will work.
Two things must be explained here. First of all the notion of “simplest”. It is primarily the guideline for choosing between alternatives. Usually a developer considers several alternative solutions. In XP, a developer chooses the solution that is easiest to implement. In other words, the solution that fits best into the existing system, that is easiest to understand etc.
The other thing to note is that the implementation to be chosen should “possibly work”. This means the developer has to be pretty sure that it works, but there is no need to prove it. The tests are there to back up the assurance. They will show when the solution does not work. Of course, good tests are necessary for this to work.
During Refactoring, a developer could find a piece of code for which he or she has a potentially simpler solution (in the meaning given above). It is the developer’s duty to implement this solution. If it works (test result), the system will “gain” simplicity.

You Are Not Gonna Need It (YAGNI)

The other important design rule of XP is YAGNI.
During the implementation of tasks often possibilities arise for adding features that could be needed later on. In XP developers resist this temptations although it would be easy now and perhaps more difficult later to implement the feature. There are three things to observe :

- There will be more work now because there is the task at hand and the not-yet-needed feature too.
- It may well be that the feature will not be needed later, for instance because the customer changes a requirement.

- If the feature is implemented now it will need maintenance (Refactoring, tests etc.) and additional communication, and there is a good chance that it will be changed before it is even used.

Of course it seems to be cheaper to add the feature now than to add it later. XP advocates state that in XP you won't gain much, since it relies on "Do The Simplest Thing That Could Possibly Work" and Refactor Mercilessly.
Another implication of this rule is that anything that is not used (anymore) should be removed. While Refactoring, methods or even classes that are supposedly not doing anything useful will be spotted. The developer removes them and then runs the tests. If they run the assumption was right – the feature had no function (anymore). If not the developer will undo the removal.

Refactor Mercilessly

A **Refactoring** is a behavior-preserving transformation applied to the system. It usually involves changes to names, to the ordering of code units (methods, classes) and to their dependencies. These transformations are backed up (as usual) by the unit tests. Note that behavior here means external behavior, e.g. subsystem functionality. Refactorings will create more classes with less responsibilities (less fields and methods) that are distributed in a clearer fashion.

Consider for example the **Extract Class** Refactoring (taken from [Fo99]). It is used for classes that have responsibilities which should be performed by separate classes because they are not logically connected.
The class Person stores the name of a person as well as the phone number and area code. In order to separate the responsibilities in this slightly contrived example, a developer would follow the following steps :

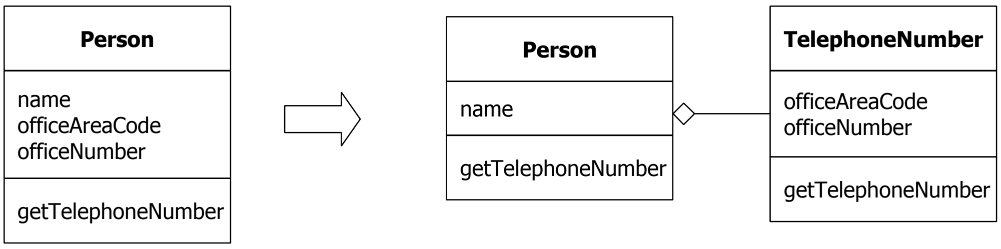


Figure 5 : Extract Class Refactoring

1. First it must be decided how to split the responsibilities. The phone number and the area code together with the getTelephoneNumber method will be moved to a new class TelephoneNumber.
2. Next the new class is created, in our case the TelephoneNumber class.
3. Make a link from the old to the new class (and back if necessary).
4. Use the **Move Field** Refactoring on every field (officeAreaCode and officeNumber) that should be in the new class. During this Refactoring every access to the fields (for instance in the getTelephoneNumber method) is replaced by a call to an accessor method called on the link.
5. Compile and test after each application of the Move Field Refactoring.
6. Use the **Move Method** Refactoring to move methods (getTelephoneNumber) to the new class. During this Refactoring every call of the moved methods would be replaced either by calling it directly on the new class or by using a wrapper method (as in our case) which simply calls the moved method. The decision on whether to use a wrapper method largely depends on the visibility of the method and whether it is used outside of the old class.
7. Compile and test after each application of the Move Method Refactoring.
8. Review and reduce the interface of both classes. If necessary change the name of the old class to better reflect its purpose.

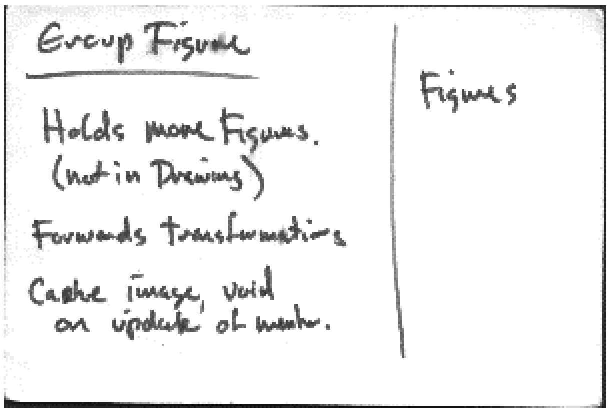
XP uses Refactoring to enhance the design of a system such that it is easier to understand and modify without affecting the behavior. If a Refactoring would not enhance the system, then it is not applied. Furthermore, Refactoring is only done as part of the development cycle: before and after the implementation of a task (before implementation to ease the integration of the new code).

Mercilessly means that if there is a chance to simplify the structure, Refactorings should be applied. Thus, Refactoring only works well when combined with Relentless Testing because it ensures that the changes due to

Refactoring do not affect the behavior. Other necessities are Collective Code Ownership and Continuous Integration.
Refactoring can have a slightly negative effect on performance, but the code tends to be better suited to optimization later on (see Lazy Optimization).

CRC Session

CRC stands for **Class Responsibility Collaborator**. It is a method to determine and visualize the design of a system.
The main tool for this method is the index card. Each card represents a class within the system.
During a CRC session one starts with a set of cards onto which the names of obvious classes are written (at the top). If a responsibility (e.g. field or method) is found it is written on the left side of the card. If the class needs to cooperate with another class to fulfill a responsibility a collaboration exists. The name of the other class is then written on the right side of the card.
The cards can also be of different colors to express different types of classes.



Picture 6 : Example of a CRC card

More important than the responsibilities and collaborators is the layout of the cards. Cards that have a tighter relationship (for instance composition) are placed nearer to each other. The placing of the cards expresses the structure of the system at hand. Moving the cards around can be used to try out different “designs”.
The design is tested with a “walk” of User Stories (or Engineering Tasks). In order to fulfill the story the cards are examined step by step along the collaboration hierarchies.

XP utilizes CRC cards as a dynamic discussion technique and not as a documentation technique. The cards are used as props in determining the interaction between classes and only a couple of cards are used at a time. Usually the cards are not preserved because it is quite easy and fast to recreate them if necessary.

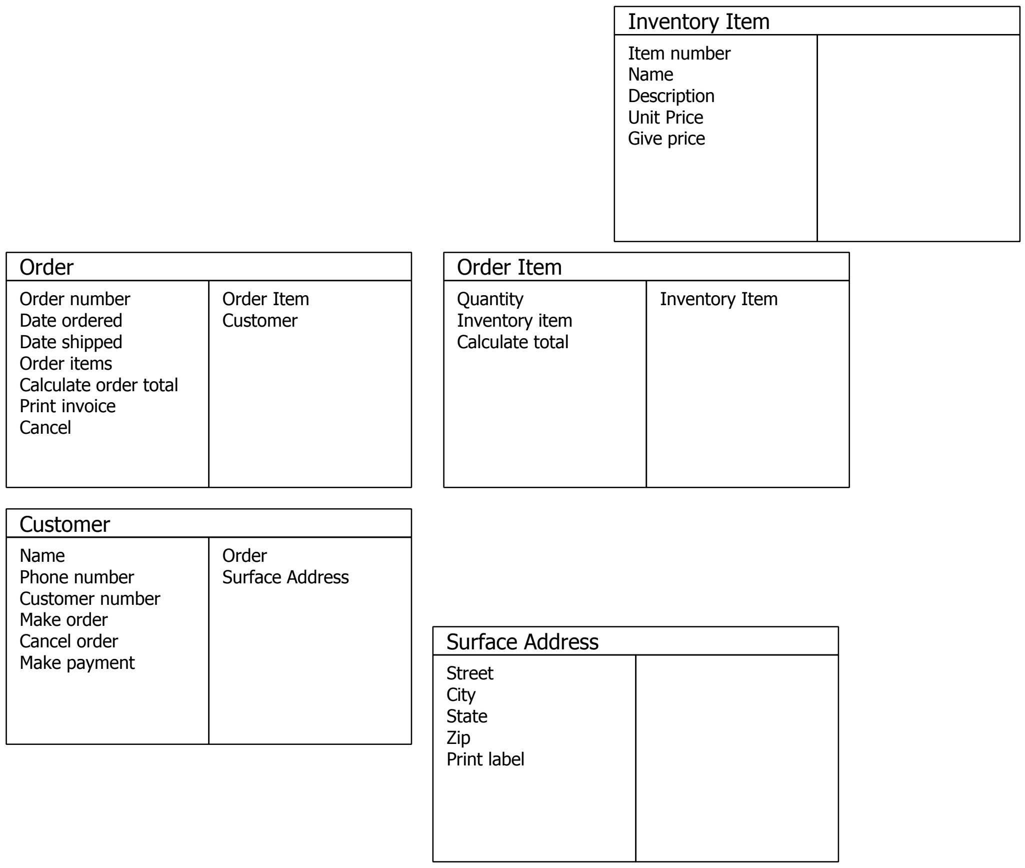


Figure 6 : Example of a CRC layout

System Metaphor

Systems are build around a single or a small set of cooperating metaphors. These metaphors are used as a basis for the system's design as well as a guiding scheme for naming classes, methods etc. The [C3] project for instance chose a manufacturing metaphor for their payroll system. It consisted of lines, parts, bins, jobs, and stations, all of which were pretty natural to a car manufacturer. With this they could rely on a rich domain model which helped in understanding the complex domain of a payroll system.
The metaphors are also important to improve communication. If they are well chosen, the customer and the developers can relate to them (business metaphors). Determining whether some functionality is already implemented is easier since one would simply ask "how would it be named ?" and then see whether a part with the possible names is there.
Therefore the metaphors should be selected carefully. If there is a situation where the metaphor doesn't fit it should be enhanced or even replaced with a new one. The replacing of a metaphor is a big step which will need a lot of Refactorings and perhaps even a new Planning Game.

Lazy Optimization

Optimizing comes as late as possible ("Make it work. Make it right. Make it fast."). Furthermore one should never try to guess the systems bottlenecks but measure them.

Conclusion

The power of XP lies in the cooperation of the various rules and practices. I've had difficulties understanding the dynamics of this cooperation from only reading about it. It simply seems as if the whole cannot really work. Given the success stories of XP projects it must work, however. I think that is necessary to try XP in an actual project in order to fully understand and evaluate it.
So far I could only try to of the practices of XP, namely Refactoring and Unit Testing.
From what I've learned, I can say that Refactoring is a powerful tool to simplify systems. Nonetheless it must be handled with care. Because you can easily destroy the system it is necessary to use tests to back it up. Furthermore it is absolutely necessary to have a good communication in the team because the system will be in a somewhat flowing state.
Unit tests are a great way to ensure that the code is doing what you want it to do. It gives you real confidence if you're able to say "this method is right because the tests run". Something interesting happens when you write the tests first. You express your expectations in code and then you really implement just enough to meet these expressed expectations. Although this moves the problem of false assumptions only from the unit to the tests, they are more easily spotted because they are explicit in the expected outcome and not implicit in the algorithms you chose. It is also easier to define the interface when you make explicit on what the method/class should work and what the results are. Unit testing is nonetheless a skill that must be learned and trained and it requires discipline. It is also hindered by language "features", pressing schedules, system barriers (for instance database systems) etc.
I therefore would say that eXtreme Programming is interesting in at least three ways :

- It is different. Although the rules and practices are not new they are used in a much more strict fashion. This is especially true for Pair Programming, Refactoring and Unit Testing.
- It seems to be more fun than a "traditional" process, mostly because there is a strong emphasis on team work, communication and helping each other.
- During XP projects every team member learns a lot. In fact one of the key benefits of XP lies in the distribution of knowledge throughout the team.

References

Books

Abbreviation	Reference
[Be99-1]	Kent Beck, “eXtreme Programming explained – Embrace Change”, Addison-Wesley, 1999
[Fo99]	Martin Fowler, “Refactoring – Improving the Design of Existing Code”, Addison-Wesley, 1999
[BeSu97]	David Bellin, Susan Suchman Simone (Contributor), Grady Booch (Editor), “The CRC Card Book”, Addison-Wesley, 1997

XP Web-sites

Abbreviation	Reference
[C2]	Wiki Wiki Web, http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
[XPOrg]	ExtremeProgramming.org http://www.ExtremeProgramming.org
[XPMag]	XP Magazine http://www.Xprogramming.com
[XPD]	XPD, http://www.xdeveloper.com
[Wake]	William Wake, “Software Design and Development”, http://users.vnet.net/wwake/

XP Articles

Abbreviation	Reference
[OO]	OOTips: Extreme Programming, http://ootips.org/xp.html
[C3]	Chrysler Goes To “Extremes” – Case Study, Distributed Computing, October 1998 http://www.xprogramming.com/publications/dc9810cs.pdf
[Be99-2]	Kent Beck, “Embracing Change with Extreme Programming“, IEEE Computer, Vol.32, No.10, October 1999, http://dlib.computer.org/co/books/co1999/pdf/rx070.pdf
[Be99-3]	Kent Beck, “Extreme Programming – Flatten the change-cost curve by using XP in project planning and testing”, C++ Report Feature Story, May 1999 http://archive.creport.com/9905/html/from_pages/feature.shtml
[Dyson]	Paul Dyson, “Does software need Architecture – or something more Extreme ?”, http://www.dyson.force9.co.uk/arcxp.doc
[RoSc]	Doug Rosenberg & Kendall Scott, “XP – Cutting Through the hype”, Objective View, Issue 3 http://www.ratio.co.uk/ov3pdf.zip
[Wind]	Mark Windholtz, “Software Development Best Practices”, http://w3.one.net/~objwind/present/ExtremePractices.htm

CRC Articles

Abbreviation	Reference
[Ambler]	Scott W. Ambler, “CRC Modelling – Bridging the Communication Gap Between Developers and Users”, http://www.ambysoft.com/crcModeling.PDF
[BeCu89]	Kent Beck & Ward Cunningham, “A Laboratory For Teaching Object-Oriented Thinking”, OOPSLA’89 Conference Proceedings http://c2.com/doc/oopsla89/paper.html
[Bjork97]	Russell C. Bjork, “An Example of Object-Oriented Design – An ATM Simulation”, http://inprem.rug.ac.be/~gpremer/OOA/ATM_Example/
[Brum96]	Nils Brummond, “Object Oriented Analysis and Design using CRC Cards”, 1996 http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/crc_b
[Coc99]	Alistair Cockburn, “Using CRC Cards”, Humans and Technology Technical Memo HaT TR.99.01, March 1999 http://members.aol.com/humansandt/techniques/crc.htm
[Rubin98]	David M. Rubin, “Introduction to CRC Cards”, Softstar Research, 1998 http://www.softstar-inc.com/Methodology/CRCIntro.htm

Refactoring & Testing Articles

Abbreviation	Reference
[BeGa]	Kent Beck & Erich Gamma, “Test Infected – Programmers Love Writing Tests”, http://members.pingnet.ch/gamma/junit.htm
[Jeff99]	Ronald E. Jeffries, “eXtreme Testing – Why aggressive software development calls for radical testing efforts”, Software Testing & Quality Engineering, March/April 1999 http://www.xprogramming.com/publications/SP99%20Extreme%20for%20Web.pdf
[Opd92]	William F. Opdyke, “Refactoring Object-Oriented Frameworks”, Ph.D. Thesis, University of Illinois, 1992, ftp://www.laputan.org/pub/papers/opdyke-thesis.pdf
[RoBrJo]	Don Roberts, John Brant, Ralph Johnson, “A Refactoring Tool for Smalltalk”, University of Illinois, http://st-www.cs.uiuc.edu/~droberts/tapos/TAPOS.htm

Programming In Pairs Articles

Abbreviation	Reference
[HaMa99]	David Harvey & Peter Marks, “New Cultures of Programming”, JaCC Conference 1999, http://www.ftech.net/~honeyg/articles/Culture-dist.zip
[WiKe99]	Laurie A. Williams & Robert R. Kessler “The Effects of ‘Pair-Pressure’ and ‘Pair-Lerning’ on Software Engineering Education”, http://www.cs.utah.edu/~lwilliam/Papers/CSEET.PDF
[CoWi00]	Alistair Cockburn & Laurie Williams, “The Costs and Benefits of Pair Programming”, http://www.cs.utah.edu/~lwilliam/Papers/XPSardinia.PDF
[WiKe]	Laurie A. Williams, Robert R. Kessler, Ward Cunningham, Ronald E. Jeffries, “Strengthening the Case for Pair-Programmin”, http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF

Other

Abbreviation	Reference
[Coc]	Alistair Cockburns Homepage http://members.aol.com/acockburn/
[Martin99]	Robert C. Martin, “Iterative and Incremental Development”, Parts I, II and III C++ Report, Engineering Notebook Column, June 99 http://oma.com/PDF/IID%20I.pdf http://oma.com/PDF/IID%20II.pdf http://oma.com/PDF/IID%20III.pdf
[Wi]	Laurie A. Williams Homepage http://www.cs.utah.edu/~lwilliam/