

## HOVEDOPPGAVE

---

**Kandidatens navn:** Per Christian Nødtvedt

**Fag:** Datateknikk

**Oppgavens tittel (norsk):** En kombinasjon av Rational Unified Process og Extreme Programming

**Oppgavens tittel (engelsk):** Combining the Rational Unified Process and Extreme Programming

**Oppgavens tekst:**

In modern software development, “Time to market” is gaining importance. Several methods to address this fact have been developed. The last decade has seen the acceptance of incremental methods. The Rational Unified Process (RUP) takes the top-down approach with increments from 1 to 3 months, while Extreme Programming (XP) is bottom-up with increments from 1 to 4 weeks.

The thesis will see whether these two methods can be combined. The results of a theoretical treatment will be tested in a case study. The work can be based on earlier work by S.F. Andersen and P.C. Nødtvedt.

---

Oppgaven gitt:	20. januar 2002
Besvarelsen leveres innen:	17. juni 2002
Besvarelsen levert:	17. juni 2002
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veileder:	Reidar Conradi

Trondheim, 17.juni 2002.

Reidar Conradi  
Faglærer



## **Abstract**

The last decade has seen a considerable advance in software engineering, and the spawning of a host of new methods. Of these methods, the Rational Unified Process and Extreme Programming enjoy considerable popularity today. They are perceived as contrasting, as they focus on radically different areas of the software engineering discipline.

Nonetheless, it would be interesting to combine the two methods, as their respective strengths and weaknesses complement each other. This thesis will present a theoretical discussion on two possible combinations of the methods. One approach is a radical fusion of the two methods. The other is more conservative where elements from XP is introduced in RUP. From this discussion, a case study quantifying whether the combination is useful has been conducted.

The case study employed use case point calculation in a novel way. We attempted to calculate defect density with use case points. The method proved to be immature for the approach taken in this thesis.

The thesis concludes, on very limited evidence, that the project under study had no quantitative improvements from the combination. Qualitative evidence show that the combination is interesting, and further work on the theme is advisable.



## Preface

This graduate thesis was written at NTNU's Department of Computer and Information Science (IDI). The thesis is a part of the Software Engineering group's work on incremental- and component-based development. The larger context of the project is the INCO project, a collaboration between the University in Oslo and NTNU.

The author would like to thank his advisor Reidar Conradi for valuable contributions during the graduate work. Tor Stålhane willingly shared his experience on the GQM method, which greatly improved the quality of the research chapter. In addition, Gunnar Nordseth, Skule Johansen, Sigurd Stendal and Øystein Fiplingdal at Mogul offered their time and enthusiasm for the case study. Lastly, the anonymous reviewers and employees at Mogul eased the thesis towards a successful conclusion.

Trondheim, June 2002

Per Christian Nødtvedt



# Contents

- 1 Introduction 3**
  - 1.1 Thesis context . . . . . 3
  - 1.2 Thesis structure . . . . . 4
  
- 2 The Rational Unified Process 5**
  - 2.1 Overall structure of the Rational Unified Process . . . . . 5
  - 2.2 The Rational Unified Process Model . . . . . 8
  
- 3 Extreme Programming 10**
  - 3.1 A “philosophical” overview . . . . . 10
  - 3.2 Development techniques . . . . . 13
  
- 4 Combining XP with RUP 16**
  - 4.1 Unifying points in XP and RUP . . . . . 16
  - 4.2 Common points in RUP and XP . . . . . 17
  - 4.3 Main differences between XP and RUP . . . . . 18
  - 4.4 Summary of the comparison . . . . . 19
  - 4.5 XP as a RUP development case . . . . . 20
  
- 5 Introducing XP elements in RUP 23**
  - 5.1 A strategy by levels . . . . . 23
  - 5.2 An example process . . . . . 24
  
- 6 Research Method 27**
  - 6.1 Improving the Rational Unified Process . . . . . 27
  - 6.2 Detailing the GQM abstraction sheet . . . . . 27
  - 6.3 Metrics on variation factors . . . . . 29
  - 6.4 Case study context . . . . . 31
  
- 7 Case study results 32**
  - 7.1 Results . . . . . 32
  - 7.2 Discussion on case study results . . . . . 34
  - 7.3 Discussion on the project context . . . . . 37
  
- 8 Case study conclusion 40**
  - 8.1 Conclusion . . . . . 40
  - 8.2 Suggestions on further work . . . . . 41

<b>A Use case point calculation</b>	<b>42</b>
A.1 Method for computing use case points . . . . .	42
A.2 Use case point calculation for the reference project . . . . .	43
A.3 Use case point calculation for the case project . . . . .	46
<b>References</b>	<b>49</b>

# Chapter 1

## Introduction

Over the last decades, software development processes have been intensely studied. Organisations delivering software are continuously attempting to improve their work by using different methodologies. Some methodologies are short lived, while others have a tendency to survive even though their value is doubtful.

This thesis will take a closer look at two competing methodologies developed these last few years. The Rational Unified Process is a top-down framework for making development methodologies, while Extreme Programming is a bottom-up approach based on programming as the critical activity.

The goal of the thesis is to test a combination of the two methodologies. Is there a quantifiable difference between a purely RUP based process and a process that blends RUP with XP?

The thesis attempts to answer this question with a case study conducted at a Norwegian software development firm. The rest of this chapter will present the context of the thesis, and the structure of this report.

### 1.1 Thesis context

The graduate thesis was written at the Norwegian University of Science and Technology (NTNU), Faculty of Information Technology, Mathematics and Electrical Engineering. The thesis is a part of the Software Engineering group's INCO project at the Department of Computer and Information Science. INCO is a cooperation between the computer science departments at the University in Oslo (UiO) and NTNU respectively. It is backed by the Research Council of Norway.

The background for INCO is the high cost and high failure rate of software projects today. Some of the proposed solutions to these problems include incremental development and component-based development. These methods can help reduce development time and cost, and increase the quality of the developed software.

Although incremental development and component-based development have been known in the industry for some years, these fields of software development are still immature. The main problems lie in the area of planning and estimating cost and risk. The overall goals of the INCO project [INC] are therefore:

- Advancing the state-of-the-art of software engineering, focusing on technologies for incremental and component-based software development.
- Advancing the state-of-the-practice in software-intensive industry and for own students, focusing on technologies for incremental and component-based software development.
- Building up a national competence base around these themes.
- Disseminating and exchanging the knowledge gained.

This thesis attempts to address the first and second of these goals, in that it introduces a new variation on existing software development practices. Since the thesis is a case study, it will hopefully allow the team under study to advance their own knowledge in the field of software engineering.

## 1.2 Thesis structure

The rest of the thesis is structured as follows:

- Chapters 2 and 3 describes the Rational Unified Process and Extreme Programming.
- Chapter 4 discusses a theoretical combination of the two methodologies, and argues that they are, in essence, one and the same.
- Chapter 5 presents a less radical approach to combining the methodologies compared with chapter 4.
- Chapter 6 presents the research method used in the thesis, along with the hypotheses used in the case study.
- Chapter 7 presents and discusses the quantitative and qualitative results from the case study.
- Chapter 8 draws a conclusion on the discussion in chapter 7. It also discusses possible avenues for future work founded on this thesis.

# Chapter 2

## The Rational Unified Process

The Rational Unified Process (RUP) [Kru99] is a process framework developed by Rational. It is based on the Unified Software Development Process (USDP) [JBR99]. Its goal is to provide a flexible framework for developing iterative information system development processes. The framework is a product, undergoing continuous development.

This chapter will give a brief introduction to RUP, discussing the most important properties of the framework. Certain expressions particular to the process will be introduced and explained, while at the same time striving to give an overall image of what RUP provides.

### 2.1 Overall structure of the Rational Unified Process

The superstructure of the Rational Unified Process is the development cycle. A project can consist of any number of these cycles, and an evaluation on the continuation of the project is made at the end of each cycle.

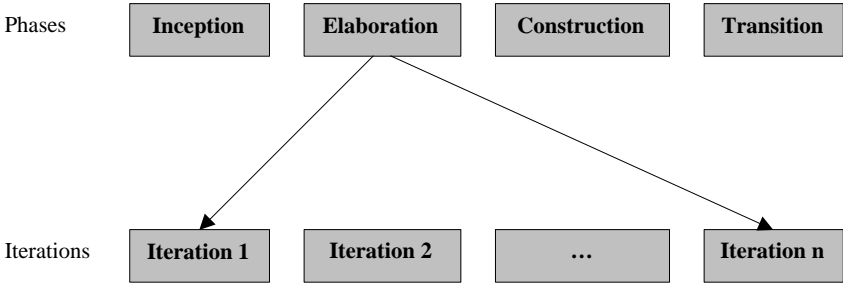


Figure 2.1: RUP's development cycle.

The cycles are divided into four phases which again are divided into iterations. The number of iterations in a phase is not specified by the framework, and will vary from project to project. Figure 2.1 shows this structure with the four phases, and the iterations of the elaboration phase. Depending on the phase, iteration contents will change.

Figure 2.2 shows a birds-eye view of RUP. It shows the four main phases of the process cycle, and the different disciplines that are exercised during the cycle. Note that the notion of disciplines replaces the workflows of earlier versions of RUP.

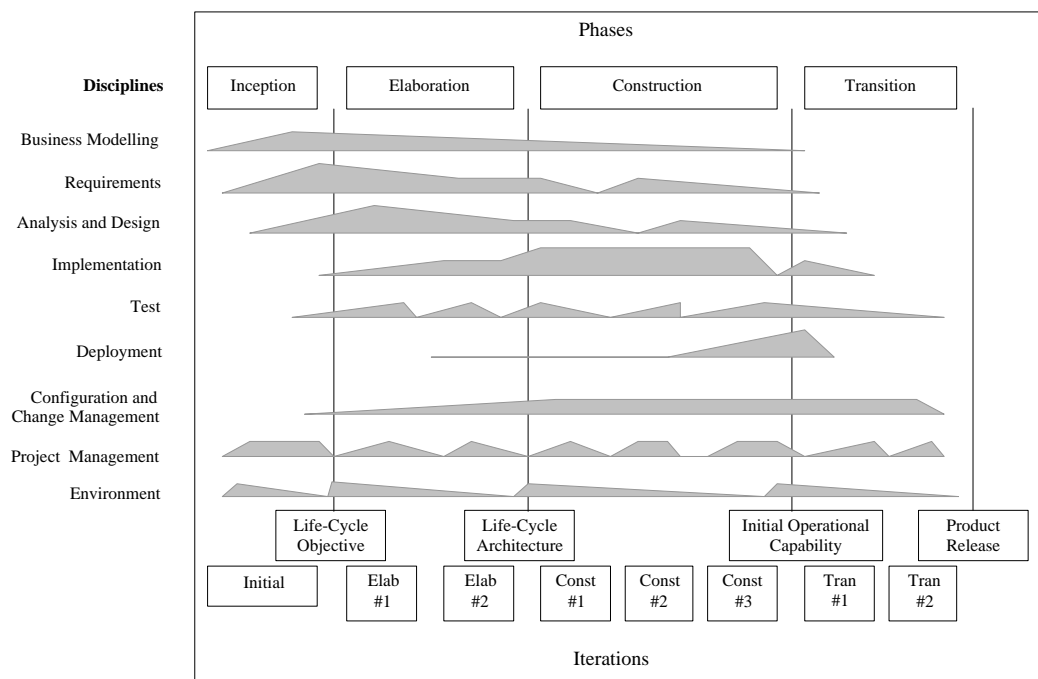


Figure 2.2: Disciplines in the Rational Unified Process.

The process is configurable, and an application of RUP is called a development case. These development cases are supposed to take into account the peculiarities of the local environment. RUP will continuously change in a company, depending on variations of the project context.

The following subsections will discuss the notions of phases and their milestones and iterations. Thereafter follows a description of the basis for all development in RUP, the use case, and lastly RUP's use of components.

### 2.1.1 Phases

The work in the Rational Unified Process' project cycle is divided into four distinct phases (figure 2.2).

**Inception:** During inception, the goal of the process is to explore new ideas. The inception phase is a pre-project. When this phase ends, a decision is made whether the project should be stopped, or if further funding is economically viable. Important disciplines in this phase includes business modeling and requirements analysis. The business modeling aims at a detailed understanding of the system's context, and requirements analysis captures the user's demands of the system.

**Elaboration:** After the initial work on the inception of a project, the elaboration will go into further detail. The phase contains detailed analysis and design, and continues requirements analysis and prototyping.

**Construction:** As the name suggests, this phase contains the actual implementation of the system. Builds and support for different platforms are prepared for deployment.

The discipline called configuration and change management is concerned with the life-cycles of the individual components in the system.

**Transition:** Nearing the end of a project cycle, the system will be thoroughly tested, and finally released. This phase allows for debugging of the system, and polishing the release in the deployment discipline. The goal of this phase is to have a finished product ready for release to the customer.

How do we know when a phase is finished and the project ready to advance? RUP answers this question with a notion called milestones. A milestone defines certain standards that should be met before a project can advance, thereby creating a measure on progress. The following list presents the milestones:

**Life-Cycle Objective:** The milestone that marks the end of the inception phase. This milestone will contain definitions on scope, estimates and a preliminary requirements specification.

**Life-Cycle Architecture:** When the use case model of the system is done, the architecture is stable and a prototype is running, the system has reached the milestone that leads to the construction phase.

**Initial Operational Capability:** To reach this level of maturity, the product must be in beta release, allowing it to be distributed to users. This demands that user support infrastructure and documentation is in order, and that the user platform has a satisfactory integration of the product.

**Product Release:** Based on user feedback in the deployment phase, changes to the system have to be implemented. When all these changes are accounted for, the system is ready for release and the final milestone in the project cycle has been reached.

A project does not necessarily end even though it has finished its transition phase. A new inception phase can be undertaken to see whether any large changes are necessary, and further phases may follow based on this decision. A project may go through several such cycles before it is shut down, thus the process provides an overall iterative pattern for managing software projects.

### 2.1.2 Iterations

Each phase in the process can be split into iterations. The goal of these iterations is to provide small improvements and continuous feedback. This way, the project's artifacts (see section 2.2.4) will be gradually refined and improved until a satisfactory level of quality has been reached.

The number of iterations in the different phases will vary from project to project, but the basic and obvious rule is that one should take the number of iterations necessary to reach the next milestone. If unfinished tasks are postponed into the next phase, team member's morale may drop. Always consider the advantages of rapidly advancing through the phases compared with having baggage from the earlier phases towards the end of the project.

### 2.1.3 Use Case based development

For simplicity, the development in RUP focuses on the use case as the basic modeling construct. These diagrams are simple and compact, and are able to convey information to people without technical background. This is an obvious advantage since it allows designers to communicate with both management and development personnel using the same diagrams.

The use case diagrams also forms a strong link to the Unified Modeling Language (UML) [Fow99b, BRJ99]. This language is in the process of standardization, leading to a useful tool for inter-project communication. Another advantage to using UML is its roots in object-oriented development. This leads to strong support for this programming school, to the detriment of other philosophies like functional- and logical programming.

### 2.1.4 Component based development and design

In [Kru99], component based development is cited as an important aspect of RUP. During architectural development, one should consider using standardised components and incorporating existing modules into new projects. [Kru99] says:

A definition of *component* must be broad enough to address conventional components (such as COM/DCOM, CORBA, and JavaBeans components) as well as alternative ones (Web pages, database tables, and executables using proprietary communication). At the same time, it shouldn't be so broad as to encompass every possible artifact of a well-structured architecture.

The use of components in development is one of the best practices advocated by RUP. Use of components does not modify the process in any particular way, although certain development cases (section 2.2.5) can be derived from the use of components in development and design.

## 2.2 The Rational Unified Process Model

To define the activities in the different iterations and phases, as they are envisioned in figure 2.1, the Rational Unified Process defines certain basic building blocks. These are all modeled in use cases, so that their description is formalised. This section will explore the notions of disciplines, roles, activities and artifacts in greater detail. Lastly, the local customizations, called development cases, are presented.

### 2.2.1 Disciplines

A discipline is a series of tasks done by a group of roles. The disciplines are the level of abstraction just below the phase-level, and all disciplines contain one or more use case diagrams. These diagrams show the order of activities within a discipline. The discipline also states who performs the activity. Different disciplines demands different personnel and deliverables, and lists of roles and artifacts are given.

### 2.2.2 Roles

Within a discipline, one must always specify who will perform a certain task. RUP defines different roles. A role is a person fulfilling a function. Although one person may fill a specific role in one iteration, another person may play the same role in another.

One person may wear different hats during an iteration or phase, acting as both Test Designer and Tester during the Test Discipline, for example. This ensures maximum flexibility towards personnel management, and few restrictions on who gets to do what.

### 2.2.3 Activities

Within a discipline, several different activities take place. These activities detail the work that the different roles do, and the activities are specified in relation to each other. During testing for example, one must necessarily plan the tests before they are implemented or evaluated.

The activities and their interrelations are modeled as UML activity diagrams, showing the flow of work within the discipline. These diagrams function as a fine-grained plan of an ideal discipline.

### 2.2.4 Artifacts

The results of the process of disciplines, and their related concepts of roles and activities, are the artifacts. An artifact is a piece of code, a diagram, a test or anything that can convey information without direct verbal communication. They are the trace of the system, and some of the artifacts are constantly refined throughout the phases of the project.

The project manager will decide if an artifact should be produced or not, as not all artifacts will provide value to a given project. One must therefore not blindly churn out any and all elements prescribed by the disciplines just to be able to say that one follows RUP. Rather, the artifacts specified in the disciplines serve as guidelines, building on best practices.

### 2.2.5 Development cases

RUP is not a blueprint on how software projects should be executed. Each project has its own challenges and peculiarities, and imposing a set model in all situations cannot succeed. Therefore, RUP has development cases.

A development case is a modification to the process to suit a specific need. An organisation should and will tune the process to suit their particular demands, and this is encouraged by the framework. These customizations can appear at any level, and examples include adding an extra phase, dropping a discipline, introducing activities and artifacts in a discipline and so forth.

This flexibility ensures that the framework is useful for many different types of software development projects. A weakness to this approach is that it is complicated for new users to pick out the gist of the process, thereby creating a steeper learning curve.

# Chapter 3

## Extreme Programming

Extreme Programming (XP) is a collection of lightweight development techniques intended to provide software development teams with a minimal but structured approach to development. It was introduced in [Bec00], and has since spawned a sizeable user community and a lively debate.

This chapter will give short descriptions of the most important practices in XP, and will also discuss the peculiarities of the methodology.

### 3.1 A “philosophical” overview

This section will discuss the overall theories and approaches to managing and implementing an Extreme Programming project.

#### 3.1.1 The values

XP is founded on four basic *values*: communication, simplicity, feedback and courage. These four principles should guide the mind-set of the developers, helping them achieve a more implicit ideal of mutual respect.

An XP project cannot function if its participants ignore these principles, as the different practices used in a project are founded on these ideas. Subsections 3.1.2 through 3.1.5 discusses some practices that are derived from these values.

#### 3.1.2 Story Based Development

In contrast to RUP, XP does not prioritize modeling and up-front design. Effort is concentrated on providing value as quickly as possible. The method for arriving at this goal is story based development. A story forms the basis of the planning game (section 3.1.3), and replace the requirement documents and project plans of other development methodologies.

The customer is responsible for writing user stories that describe one feature of the system from the user’s point of view. Acceptance tests are used to verify when a story is successfully implemented. A user story can be compared to RUP’s use cases, albeit with less formality.

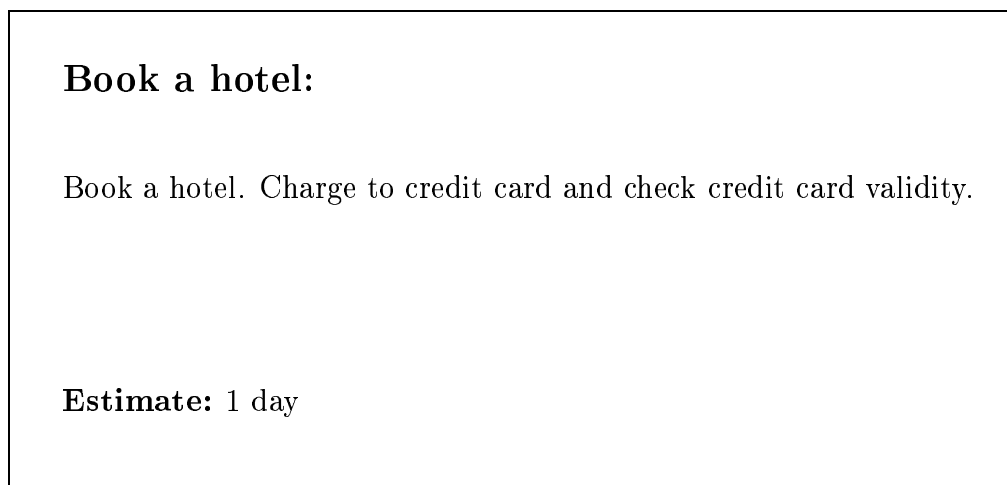


Figure 3.1: A simplified example of a story card.

The XP practice is to use a form of user stories that work for the project group. This means that their physical representation can vary from project to project. A popular way to use stories is to write them on cards for easy handling. A card can be passed around, manipulated and especially thrown away when it is no longer needed. Figure 3.1 gives an example of a story card. Note the simplicity of the text, in that it only says what the system should do, not how it should be done.

Throwing away story cards may seem odd to people that are used to storing a trace of the development and design effort. The rationale for discarding them away is that all information is encoded in the test cases and the code. Thus the cards are redundant.

The cards are used in the planning game (section 3.1.3) as objects for bartering between the customer and developers in the planning process. This practice of development is inspired by the values of communication, simplicity and feedback, in that the basic unit for communication use techniques that non-specialists can comprehend. The planning game allows the programmers to rapidly give feedback on customer’s ideas, and vice versa.

### 3.1.3 The Planning Game

XP strives for simplicity, and this ideal touches every part of a software project. The planning technique used in XP is a lightweight and simple process based on honest communication.

A planning game session is intended to arrive at a prioritized list of user stories to be implemented. The customer will write these stories, and the developers will cooperate to arrive at an effort estimate. This estimate is calculated by breaking the story into several smaller *tasks*. A task is small unit of work that can be implemented in less than a day. When a story has been completely decomposed into atomic tasks, the estimates are summed to a total for the entire story. Within the next development iteration, the customer decides which stories to be implemented, and the relative priority of the different stories.

After all estimates are done, the number of stories that are possible to implement in the next iteration is calculated. If the team has too many stories to implement, some will have to be deferred to a later iteration. The customer will choose the stories to postpone. On the other hand, if a team realize that they have too few tasks for one iteration, they can demand another user story to fill up their schedule. This honesty in communication is essential, as the developers have to both give and take to get the customer to accept adjustments in scope.

The idea is that software development is controlled by four variables. These variables influence each other. To arrive at a balance between what the customer demands and what the developers can implement, the setting of these variables is divided between the customer and developers. The variables are as follows.

- Cost
- Time
- Quality
- Scope

The customer decides the cost, quality and time, while the developers decide the scope. This means that if the customers demand a delivery at a certain date with the given team and with a given level of quality, the team gets their say on how much they can implement. If the customer demands too much, he or she will have to stop and think through what is really important, before leaving out the parts that can wait.

The aforementioned process leads to a balance between the customer and the developers, in that both get their say in what can and should be developed. XP projects do not work if a manager demands a set of features on a certain date without discussion. On the other hand, an experienced XP group will have reliable estimates, which means that the team will actually deliver what they have committed to. The practice is founded on the value of communication, and strives towards the value of simplicity. It also demands feedback from all participants, and courage from the developers to take responsibility for the tasks they estimate.

### 3.1.4 Customer Involvement

In any XP project, continuous customer involvement is critical to success. The customer, or the person representing the customer, is responsible for making all priorities among stories. If a task takes more time than anticipated, the customer can choose whether it is worth the effort to continue, or to defer the task to a future iteration. These ideas are derived from the values of communication and feedback discussed in 3.1.1.

To arrive at this level of communication, it is essential that the customer is present. If this is not possible, a project manager can act as a stand-in for the customer. The point is that a developer should always be able to ask for a clarification on a story or demand a new prioritization among tasks. The customer does not need to be a technically educated person, and experiences [Bec00] have shown that expert users who will use the finished system work best.

Customer integration on software development projects is not a new idea, as shown by the Dynamic System Development Method (DSDM) [MS95]. This process is a formal-

isation of Rapid Application Development (RAD) [Her95], made by the Dynamic System Development Method Consortium.

### 3.1.5 Planning for Change

A basic tenet in XP is that everything will change, so be prepared for it. Any decision that does not need to be taken today should be postponed until the last moment it can possibly be taken. Often, the context of the problem changes, and the decision loses its relevance. This means that important decisions are taken when they need to be taken, and this leads to simplicity. If planning for change is to succeed, the developers have to be courageous.

This idea means that everything should be made as simple as possible. A simple design solving today's problem is better than a nifty solution that solves a problem that may arrive next week. Code will be reworked through refactoring (section 3.2.3) anyhow, so unneeded complexity will always disappear. This may seem like ad-hoc programming, but with a methodical approach to testing and refactoring, it translates into providing the simplest solution as fast as possible. In XP this is a Good Thing.

## 3.2 Development techniques

In this section, a few of Extreme Programming's development techniques will be presented in detail.

### 3.2.1 The Test First Approach

A cornerstone in the XP approach is to test thoroughly. Every user story that will be implemented gets functional and acceptance tests, and these tests are compiled into a test suite that is run often to see the progress of development. Additionally, development is supported by thorough unit tests. Every part of a program has a unit test, and these are also gathered into test suites. The unit test suites are run before and after adding a new feature, to see that no parts of the existing system have been broken by the new addition.

All unit tests are written before implementation, to force the developer to consider the interface and design of the new feature. The tests are run to verify that they fail before implementation proper begins. If some of the new tests actually work before the task is implemented, it is a sign that the developer has a poor understanding of the system, or that communication has been unclear. Usually the tests fails, and work continues to extend the system until all the new tests run. At this point, the programming task is finished and the new code and tests can be integrated with the system. Thus, the code base and test suite grows in parallel, boosting confidence in the system.

In addition to the unit tests, the customer will provide acceptance tests to help decide when a story is completely developed. The developers will write functional tests to verify technical characteristics. Examples of functional tests are load tests, response time tests and so forth.

### 3.2.2 Pair Programming

Of the practices of XP, the one that is normally taken as the most radical is pair programming. This practice involves placing two developers on a single machine, and have them develop together. One person will type in the code and provide detailed solutions, while the other will have a wider perspective and focus on the overall solution.

An obvious advantage to this technique is that it provides continuous code review. Every line of code has been read by one person in addition to the person writing the code. Trivial errors are spotted earlier, saving compilation time, much less debugging time. Additionally, the person responsible for writing the code must also convince the other person that his or her solution is correct. This leads to simple solutions, and these solutions are less likely to contain errors as two people have agreed on them and seen them implemented.

Pair programming is also a way to make the test-first approach more effective. The two developers will see more ways to test a given story, and it is harder for a time-pressed programmer to forego testing when another developer is looking over his or her shoulder. Thus, pair programming reinforces the advantages of the test-first practice.

A common complaint on pair programming is that it provides less value for money to the employer, as two people working in parallel on different tasks could produce more code than two people cooperating on the same code. This argument is refuted by [WKCJ00], where it is pointed out that programming in pairs provide higher employee satisfaction, higher code quality, and after an introductory phase of lower effectivity, higher development speed.

### 3.2.3 Refactoring

Refactoring is a process that aims at two goals:

- Clarifying existing code to make it more readable
- Improve design

[Fow99a] gives an extensive catalog of different refactorings. It contains a detailed description of signs when a refactoring is in order, how to implement it, and tips on when to use (or not use) a certain refactoring. The practice aims at taking a piece of existing code and modify it so that the resulting code is “better”.

This is not a substitution of traditional up-front design, but a way of pointing out possible improvements to the programmer. These possibilities can then be taken into account by the designers.

Refactorings are not intended to change the functionality of the system, only the appearance of the code. To successfully refactor, one *must* have a comprehensive set of tests that permit the developer to see whether the change has introduced any unintended side effects. This technique therefore goes hand in hand with disciplined automated tests.

Many refactorings strive towards the introduction of patterns [GHJV95], attempting to replace poorly constructed code with standardised solutions from the pattern tool box. Thus, the programmer may influence the designers to learn and use (to them) new patterns, and to help construct more robust solutions.

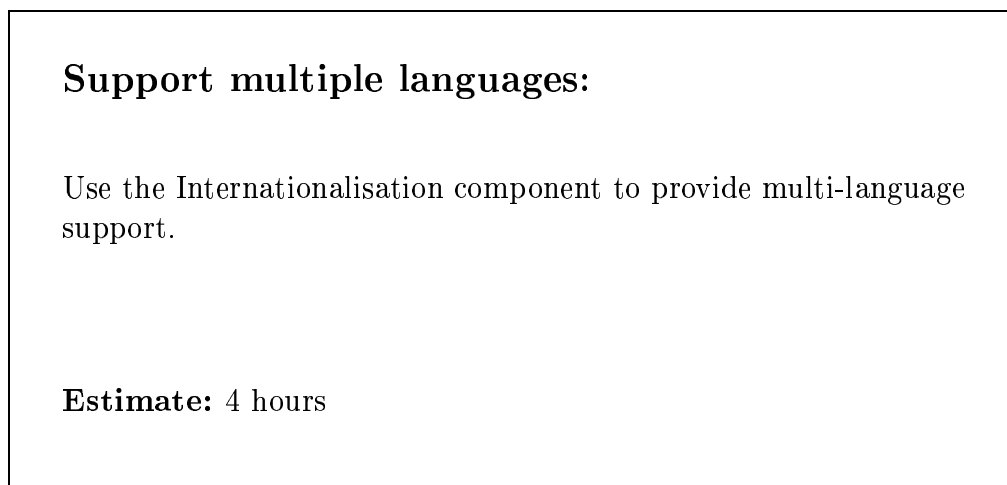


Figure 3.2: A task card using COTS.

### 3.2.4 Short release cycles

XP advocates very short release cycles, often down to a week or days. This is to allow the customers to provide rapid feedback, so that new stories can be added or story priorities changed. The first iteration of an XP project will strive to make a complete, but shallow, implementation. Thus one can rapidly get an image of what the system will look like. This will make it easier for users to provide useful change requests, and when the customer is not sure what he or she wants, this kind of of pre-releases are irreplaceable.

### 3.2.5 Use of components

One would assume the reuse of components to be an integrated part of the XP approach, but the authoritative sources remain surprisingly silent on this. Neither [Bec00], nor [BF01] mention Commercial off the shelf components (COTS) based development explicitly. On the other hand, [Pee01] point to the fact that using Enterprise JavaBeans, one kind of COTS, violates the principle of doing the simplest thing possible. Setting up an infrastructure for an existing component is usually more complicated than building minimum functionality from scratch.

Then again, XP does not dictate project context. It is possible to lay down rules governing development. One of these rules can state that components should be used where possible. During estimation in such a project, the tasks to implement a story can be based components. This way, the customer is shielded from the technical details of which component to use. Figure 3.2 shows an example of a task card where a hypothetical module for providing support for multiple languages are used.

Until a more thorough treatment on this topic surface, one can only assume that using COTS in XP projects is perfectly feasible. It is in line with the general philosophy of XP, but the details on this matter needs sorting out.

# Chapter 4

## Combining Extreme Programming with the Rational Unified Process

This chapter will discuss the possibilities for combining the Rational Unified Process and Extreme Programming. We can attempt this in several ways, but as mentioned in [EK01], the most straightforward way is by introducing elements of the XP methodology into variants of the Unified Software Development Process (USDP). The fact that RUP is a framework implies that XP can be implemented within RUP. This chapter takes the latter approach, while chapter 5 takes the former

### 4.1 Unifying points in XP and RUP

Both Extreme Programming and the Rational Unified Process claim they are based on best practices. They are built on experiences with what works in the industry, and they are therefore formalisations of how software developers operate “in the wild”.

XP does not have an explicit list of best practices, but several principles are listed in table 4.1 along with RUP’s best practices.

XP	RUP
Rapid feedback	Develop software iteratively
Assume simplicity	Manage requirements
Incremental change	Use component-based architectures
Embracing change	Visually model software
Quality work	Continuously verify software quality
	Control changes to software

Table 4.1: Principles and best practices of XP and RUP

Both methodologies point out the importance of incremental development, but XP accentuates the importance by dividing the notion into two points. *Rapid feedback* and *Incremental change* both amount to developing software through a process of gradual growth influenced by the customer. RUP has *Develop software iteratively* as its first point, thereby placing solid emphasis on this method. This implies that the common denominator of the two methodologies is the incremental approach to development.

*Quality work* of XP is also directly compatible to RUP's *Continuously verify software quality*. Thus both methodologies put great emphasis on producing quality software, something that should be expected.

Apart from this, the common points and differences of the methodologies are difficult to point out as the XP principles are on a somewhat higher level than RUP's best practices. The next sections will point out areas where the two methodologies match or are in conflict.

## 4.2 Common points in RUP and XP

This passage will present a certain number of elements of the two methodologies that are comparable. It is followed by a discussion that shows how Extreme Programming can be a Rational Unified Process development case.

### 4.2.1 Use cases and stories

RUP puts great emphasis on use case diagrams as the basic element for communication. A use case describes what the user sees in a simple notation that can both be understood by nontechnical people, and be refined into diagrams of higher detail and power by the designers. The use cases are a starting point for architectural design and effort estimation.

XP employ user stories to convey the same notions as the use cases, but without the formalised notation. User stories are recommended as a way to communicate easily, and the most widespread practice is to use story cards. These cards are easy to manipulate and modify, and their size restricts the stories to a high level of abstraction. As each story is written in a natural language by the customer, they will be understandable by anyone, and provide a basis for estimation and planning.

Clearly, the foundations of the two methodologies have a lot in common, although their physical representation and practices differ somewhat. This indicates a common ground that will allow us to construct a combination of the two processes.

### 4.2.2 Two ways of planning

The planning game of XP matches the project management of RUP, but the approaches are radically different. The RUP practice called *Control changes to software* from table 4.1 is already a part of XP's planning game. The same goes for *Continuously Verify Software Quality*.

### 4.2.3 Evolutionary development

As both methodologies are incremental, their systems will evolve gradually from initial essay to finished product. XP aims at constructing a shallow version of the system at an early stage. By shallow we mean a system that gives an image of the look and feel of the application, but lacking the full functionality of the entire system. This may be considered as an evolutionary prototype. In addition, developers are encouraged to build small throwaway prototypes to explore technical details. These prototypes, called

“spikes”, are written by one developer without the test first methodology. This is to discourage “upgrading” the throwaway prototype to production code.

RUP proposes the use of prototyping during the implementation phases of the project. These prototypes are divided into several categories, where the two main types are throwaway- and evolutionary prototypes. Evolutionary prototypes are greatly encouraged, as they entice a more stringent development. The process also cautions developers against upgrading throwaway (also called exploratory) prototypes into evolutionary prototypes.

We note an obvious similarity between these two approaches. XP has a more implicit idea of evolutionary prototyping through building a shallow version. RUP does not demand prototyping, but strongly suggests evolutionary prototyping.

### 4.3 Main differences between Extreme Programming and the Rational Unified Process

Several practices in the Rational Unified Process and Extreme Programming are similar, while others are in conflict. The most notable conflict is XP’s *Travel light*<sup>1</sup> principle compared with RUP’s love of documentation. XP eschews massive documentation, and any document that has lost its value will be discarded. RUP puts great emphasis on modeling up front, which is incompatible with the Travel light principle.

#### 4.3.1 Process focus

When comparing the two methodologies, the most striking difference is in the main focus. RUP defines itself as an architecture-centric process that concentrates on the top-down approach. The architecture is a vehicle to communicate the essence of a software project, and it is used to retain the integrity of the system during modification and evolution. Additionally, the architecture crystallises interfaces and the interplay among modules and components, thereby simplifying reuse.

XP is oriented at a bottom-up approach in that it focuses on code. The basic element of the system is the working code, and the accompanying test cases. By relying on the test cases to always run without failure, modifications to the system can be made with confidence.

#### 4.3.2 Programming guidelines

RUP makes no restrictions on how programmers fulfil their task. This is in stark contrast to XP, where the test first approach and pair programming are imposed as critical elements of the methodology<sup>2</sup>. XP suggests several coding techniques that are intended to support each other. An example of this is how unit testing supports refactoring, and how pair programming support unit testing.

---

<sup>1</sup>Don’t bring along anything that is redundant. Test cases and code makes story cards redundant. Metaphor makes architecture models redundant. Et cetera.

<sup>2</sup>[Bec00] even claims “You don’t get to choose whether or not you will write tests—if you don’t, you aren’t extreme: end of discussion.”

	XP	RUP
Process focus	Code	Architecture / Requirements
Evolutionary development	Yes	Yes
Iterative process	Yes	Yes
Planning horizon	Weeks	Months
Basic planning unit	User stories	Use case
Level of documentation	Low	High
Programming guidelines	Yes	No
Use of COTS	Not emphasised	Recommended

Table 4.2: Summary of the comparison of RUP and XP.

### 4.3.3 Planning horizon

As already mentioned, both XP and RUP are iterative processes. On a finer level, the processes differ on their planning horizons.

XP projects are planned per increment, and these increments are short compared with the increments in other methodologies. Beyond the running increment, the XP philosophy claims that uncertainty reigns, so making predictions is moot. Therefore, planning in any detail is postponed until it is within the running increment. Since project size in XP is limited, the iteration length have a “glass ceiling”. Anecdotal evidence suggests that this ceiling is around four weeks.

Projects managed with RUP often get a rough plan set out at an early stage, specifying approximately the length of the different phases. Beyond this, the number of iterations are estimated at the beginning of each phase. As the phases usually contain several iterations lasting a few weeks each, phase planning includes predictions for the next months. Single iterations are planned at the end of the previous iteration, thereby detailing the plan at the last moment. The iterations are usually longer than the XP iterations, but on small projects they may approach the planning horizon of XP. Normally though, the planning horizon is longer.

### 4.3.4 Use of off the shelf components

Although one would expect both development methodologies to embrace the use of COTS, this is not the case. RUP strongly advises that projects be built around existing components, to save time and money in development. This is in contrast to XP that does not explicitly encourage, or even mention, COTS.

## 4.4 Summary of the comparison

Table 4.2 sums up the similarities and differences of Extreme Programming and the Rational Unified Process as described in the two preceding sections. Further discussion on the combination of the two methodologies from the Rational perspective can be found in [Smi01, Pol01a, Pol01b].

## 4.5 Extreme Programming as a Rational Unified Process development case

As the Rational Unified Process is a general process template, it is natural to pose the question to see if it is possible to implement Extreme Programming within the framework of RUP. This section will try this, arguing that it is possible to follow both methodologies at the same time.

### 4.5.1 Unifying differences

From table 4.2, we see that the main point of difference between the methods is the process focus. Where RUP takes a top down approach focusing on requirements and architecture, XP takes the opposite approach and concentrates on the code. How can we get these seemingly opposite schools of thought to coincide?

Where RUP excels in requirements and architecture specifications with use cases, XP uses the metaphor. A good XP metaphor is a concise description of the system in comprehensive terms, filling the role as an overall architecture. Writing a good metaphor is difficult, comparable to modeling a good architecture. The main challenge is to find a good image that conveys the structure of the system in everyday terms, and few examples exists. The XP literature mentions some, but more would be in order.

In RUP, we will often work with a requirements specification. XP has no similar artifact, but the user stories fill the same role. The stories describe what the system will do, in the same way that the requirements document does, although their presentation is dissimilar. RUP, after all, places no restrictions on the format of an artifact. It merely suggests different techniques that have shown promise in the past. Should a team decide to present their requirements as story cards, then RUP will accommodate this.

At the different levels of planning, RUP uses different plans. The overall project plan that presents the lifetime of the project, the phase plans that introduce the different activities of a phase, and the iteration plans that specify the detailed execution of a single iteration. All these plans are replaced by the planning game in XP. As mentioned above, any RUP artifact is merely a suggestion. If a team chooses to replace the phase and iteration plans with the planning game, this will not conflict with the idea of RUP.

RUP	XP
Architecture	Metaphor
Requirements specification	User stories
Project plans	Planning game

Table 4.3: RUP elements and their corresponding XP practices.

Table 4.3 maps the three RUP terms discussed in the previous passage to the accompanying XP practices. This shows that although XP does not explicitly mention the RUP elements, they are still present in a different form.

### 4.5.2 Orthogonal practices

Several XP practices have no conflicts at all with the ideas of RUP. These techniques can be considered *orthogonal* in that they will not modify RUP, and can safely work within its framework. The most important of these are the following programming techniques.

- Pair programming
- Refactoring
- Test first
- Rapid integration

Within RUP's construction discipline, the programming actions are not specified in any detail. Therefore, the use of the preceding XP techniques poses no problem for RUP.

### 4.5.3 Similar practices

Since both XP and RUP are iterative processes, they share certain common elements. These elements ensure that the use of XP techniques within RUP proceeds without any big problems. The two most important of these are *evolutionary development* and *short iterations*.

RUP advocates prototyping during development, and distinguishes among several different forms. The recommended kind of prototyping is evolutionary prototyping where the prototype is gradually extended into the working product. This demands a consistent approach to development and quality.

XP demands rapid integration from day one, meaning that the product will gradually grow over time. This way, the system will gradually evolve from a skeleton into a full-fledged system. Although this is not pure evolutionary prototyping, the similarities are striking.

### 4.5.4 Restrictions on RUP to accommodate XP

As a final step in grafting XP into the RUP framework, we will have to place certain restrictions on the way the project work. The main restrictions are as follows:

**On-site customer:** XP dependency on continuous communication and re-prioritizations makes an on-site customer essential.

**Project size:** XP is not recommended for big projects [Bec00]. Big is not defined, but the literature suggests that groups bigger than 10 persons are “too big”. As XP relies on steady and abundant communication among all team members, one can easily imagine the massive amounts of inter-personal links that have to be maintained if 20 persons know what the other 19 does at all times.

RUP places no restrictions on the size of a project, and can be used for projects of arbitrary size. To accommodate XP in RUP, we will have to restrict the possible projects to small. Following the XP convention of using vague terms, we will not go into closer detail on the signification of small in this context.

**Collective code ownership:** Any programmer may at any time edit any piece of the system source code. This is a part of the communication values of XP, and without this restriction, refactoring will not work.

**Coding standard:** To encourage refactoring and collective code ownership, a common coding standard is enforced. A successful XP team produces code that is uniform in its presentation. It is impossible to see which programmer on such a team has written a certain part directly from the source code.

**40 hour week:** Since pair programming and test first programming is so intensive, programmers are usually exhausted at the end of a normal work day. To assure that the developers stay fresh and motivated, a 40 hour week is set as a limit. Overtime to meet a deadline during one week is permissible, but two weeks of overtime in a row is prohibited.

### 4.5.5 The problem with grafting XP on RUP

The preceding passage has shown that it is possible to combine XP and RUP, but one problem remains. This is related to XP's consistent denial of the existence of COTS. RUP recommends that all systems be built on component architectures, advocating reuse to save time and money. XP promotes The Simplest Thing That Could Possibly Work, which is usually not to accommodate a component architecture.

This would imply that XP can be called a development case of RUP in any system that does not need or rely on COTS. This seriously narrows the possible uses of XP.

### 4.5.6 A conclusion on XP as a RUP development case

All in all, it is possible to combine the two methodologies, provided that the caveats presented in sections 4.5.4 and 4.5.5 are taken into account. It is also possible to extend these principles regarding the combination of XP and RUP to other agile methodologies [Coc02]. RUP is a flexible framework for process development, and incorporating elements from other schools of thought is appropriate if it works for the team in question.

For organisations that find the XP approach too radical, we will present a light-weight approach that attempts to combine the best of the two worlds. In the next chapter, we will slightly modify RUP by introducing some of the orthogonal techniques mentioned in section 4.5.2.

# Chapter 5

## Introducing XP elements in RUP

Even though we can interpret Extreme Programming as a Rational Unified Process development case, many people will find this approach too radical. To appease their fears and at the same time striving for a fusion, this chapter will present a softer approach to the aforementioned combination.

### 5.1 A strategy by levels

We have several possibilities for combining the Rational Unified Process and Extreme Programming. This chapter will suggest a way of integrating elements from XP into RUP. The suggestion contain several levels with different commitment to the XP method.

**Level 1: Unit testing and test first.** Test first (section 3.2.1) is arguably XP's fundamental technique, and it is probably the most valuable addition to RUP. This practice includes using a framework for performing unit tests, and to regularly run a complete and automated test suite on the system. It is assumed that functional and acceptance tests are already implemented in the development process. Adding unit tests in a systematic way can further increase development speed and quality.

Test first and unit testing will in no way conflict with RUP's disciplines. Unit testing is easily incorporated into the implementation discipline, while functional tests and acceptance tests are already a part of the test discipline.

**Level 2: Pair programming.** As described in section 3.2.2, pair programming consists of two developers cooperating on writing code on a single computer. As with unit testing, this practice should not conflict with the disciplines in RUP.

It is difficult to estimate the first few projects when introducing pair programming into RUP. One must necessarily use historical data, and the first estimate with pair programming can easily be twice as big as normal estimates. This will improve over time as estimates have more data to use as support, in addition to the improved speed of pair programming.

**Level 3: Refactoring.** With unit testing in place, the developers have increased confidence in the correctness of changes to the code. This courage should be allowed to work freely, permitting the developers to modify and simplify code where they see

fit. With the infrastructure in place from unit testing, and the improved quality from pair programming, adding systematic refactoring should not be too big a step compared to the value of the practice.

**Level 4: Iteration planning.** A bigger step in integrating XP with RUP would be to replace the iteration planning activity of RUP with XP's planning game. Higher dedication from the developers is a possible advantage to this approach, as they themselves are responsible for the estimates they set out to fulfil. A problem with this approach is to find a person that can act convincingly as the project customer. A final argument against the iteration planning game is that it would conflict too much with RUP's estimation models and planning activities.

In sum, this amounts to enriching RUP with XP's approaches to coding. With this approach, the methodologies will not clash, except for level 4.

## 5.2 An example process

As mentioned, the techniques described in the section 5.1 all apply in the construction phase of the Rational Unified Process. As an example, we will look at how the construction phase can be modified with the ideas already introduced. In [Kru99] the following essential activities of the construction phase are listed:

- Resource management, resource control and process optimisation.
- Complete component development and testing against the defined evaluation criteria.
- Assessment of product releases against acceptance criteria for the vision.

All the techniques discussed section 5.1 are relevant to the second bullet point of this list. The rest of this section will make a level based development case of a RUP and XP combination.

### 5.2.1 Development in RUP

Figure 5.1 shows an example of a part of the development process in RUP. The Implementation discipline contains directions on relevant roles, artifacts and other necessary elements for implementation. The diagram shows a so-called workflow, specifying the ordering of tasks in a discipline.

Looking at figure 5.1, we note that the workflow contains two sequential actions at startup that divide the planning of the increment. The first action, Structure the Implementation Model, is intended to help divide the tasks among developers, while the second action concerns the assembly of these tasks.

After these two phases comes an undefined number of Implement Component actions, where developers take responsibility to implement the functionality specified by the Implementation Model. When a component is finished, it is introduced into a subsystem following the Implementation Model. These subsystems are in turn integrated into the

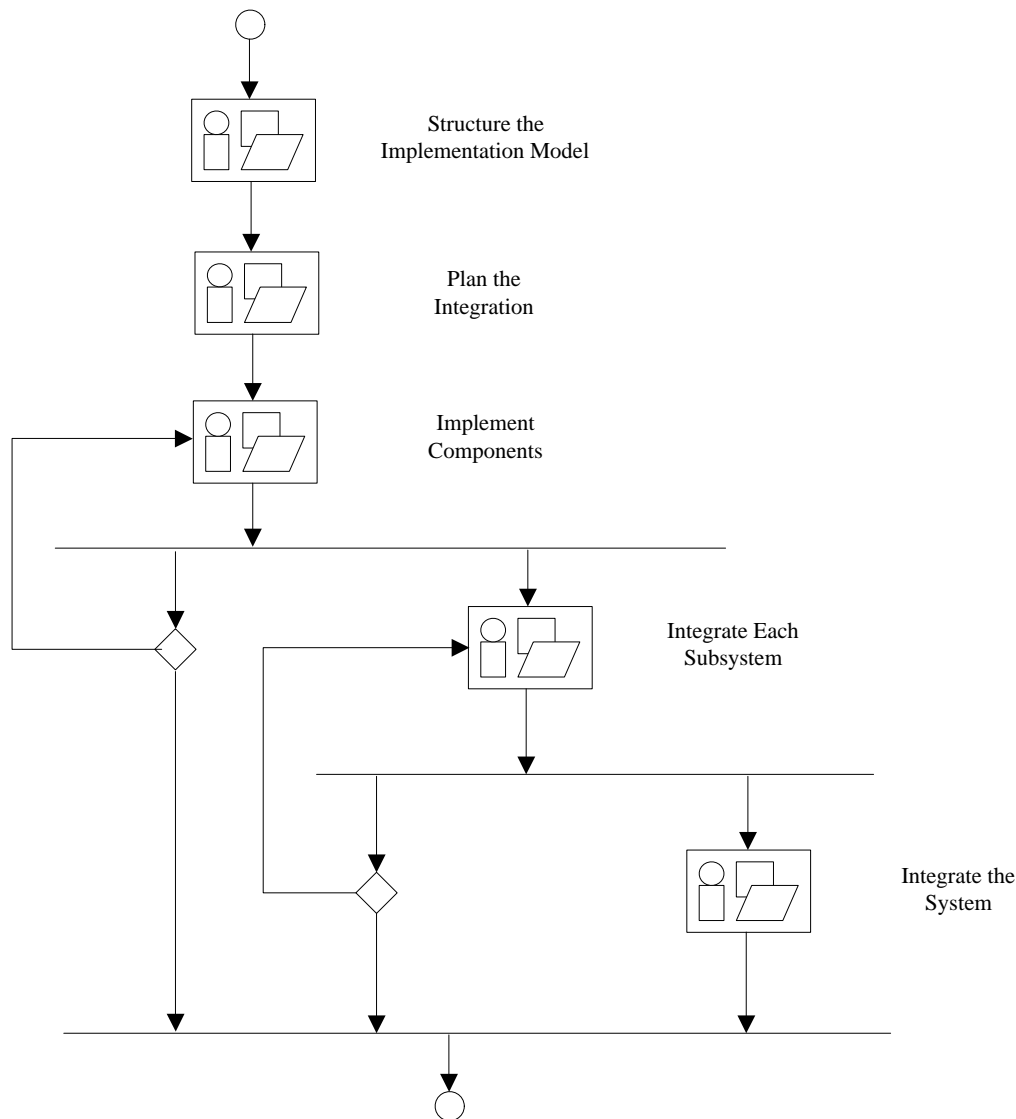


Figure 5.1: A standard implementation workflow.

total system. When all the subsystems planned for this iteration are integrated, the workflow is finished, and the project can move on to other phases or re-iterate.

### 5.2.2 The development case

Based on the levels presented earlier in this chapter, we can modify the workflow described in figure 5.1. This discussion assumes that all four levels are used, and the resulting workflow is described in figure 5.2.

We note that the start of the workflow has been simplified by collapsing the first two tasks of the standard workflow into the action called Planning Game. This action takes the role of the Integration Planning in the standard workflow, and since development has less focus on architecture in XP, the implementation model is left out altogether. One can argue whether leaving out the Implementation Model is a good idea, but the

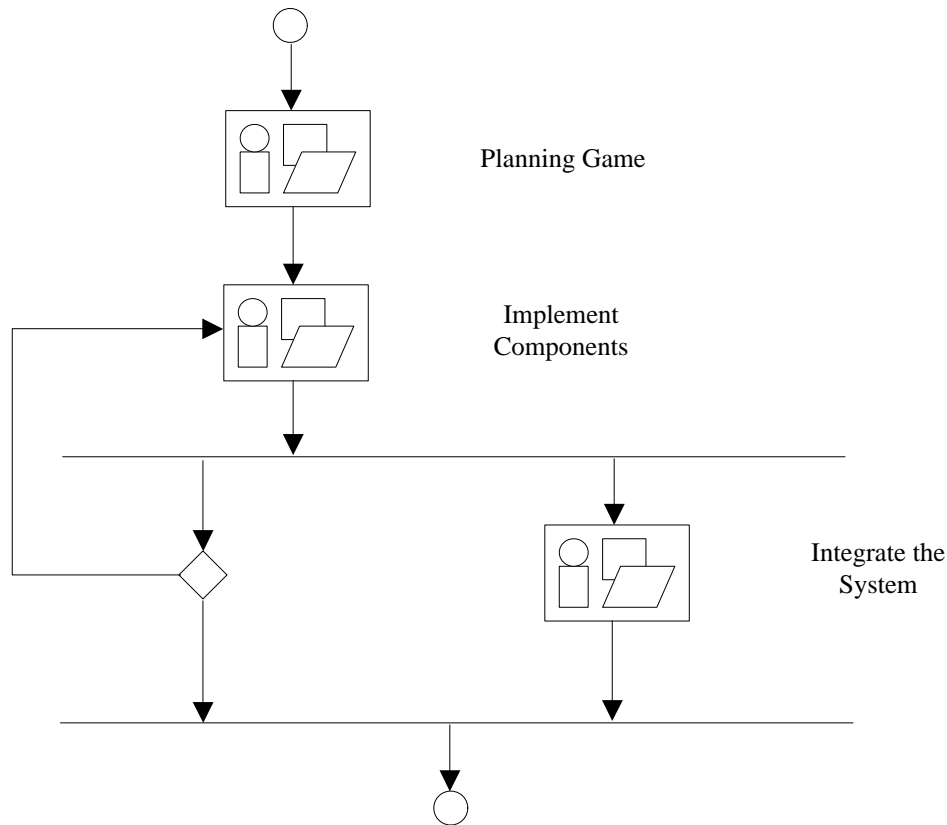


Figure 5.2: A development case based on Extreme Programming.

shared code ownership and continuous integration that is implied by pair programming and refactoring diminishes the relevance of this model.

The idea of subsystems has also been left out. This is because XP preaches continuous integration and small sized projects, and dividing a system into several distinct parts is uncommon in the XP approach. Every component developed in the Implement Component action is instantly integrated into the total system. This is XP's evolutionary system growth in practice. Using the XP practices of unit testing and automated testing reinforces rapid integration and evolutionary growth of the system.

In the case where off-the-shelf components are used, these can be integrated alongside the developed components in the Integrate the System action. The implement component action itself does not see these components, as the planning and integration of these components are incorporated in RUP's analysis and design discipline.

The levels described in section 5.1 are seemingly missing from this development case, except for the planning game. This is because RUP does not impose any development techniques, and the practices of test-first programming, pair programming and refactoring are all part of the Implement Component action. The XP practices mentioned are completely independent of RUP's framework, except for the Planning Game that replaces an existing action.

# Chapter 6

## Research Method

This chapter discusses the research method used in the project. The project was conducted as a case study. The first section of this chapter will pose the question on which the study is founded. The second will put forth three hypotheses, using the Goal Question Method (GQM). The last section will present the metrics necessary to test these hypotheses.

### 6.1 Improving the Rational Unified Process

The premise for this thesis is that the Rational Unified Process is perceived as somewhat cumbersome, and we want to modify it to become more flexible. To obtain more flexibility, we introduce elements from Extreme Programming.

To learn as much as possible of the effects of this change, we want to make measurements and try to arrive at a conclusion on whether the changes are effective or not. We have several possible research methods, but within the time-frame of this project, only a case study is feasible.

According to [Yin94], case studies are most effective when they pose questions that investigate *how* a set of actions modifies a result. We will therefore pose the following question as the basis for designing the case study:

How will the introduction of XP elements into RUP modify the results of using RUP?

This question will serve as the basis for several hypotheses. Table 6.1 shows a GQM abstraction sheet that presents the focus and hypotheses of the study. From this abstraction sheet, we can construct a measurement plan. This plan will be presented in the next section.

### 6.2 Detailing the GQM abstraction sheet

This section will be organised according to the statements given under the heading *Quality focus* in table 6.1. Each quality focus has a corresponding hypothesis, and these hypotheses are as follows:

Analyse: The development process	In order to: Check code quality improvement	Seen from: Project management	In the environment of: Mogul
<i>Quality focus</i> Q1: What is the defect density? Q2: What is the development effort? Q3: How long are the iterations?		<i>Variation factors</i> Qa: Development method. Qb: Use case properties.	
<i>Baseline hypothesis</i> HQ1: Comparable components have a defect density of 0,49 defects per use case point. HQ2: The development effort per use case is 77 person-hours. HQ3: Average iteration length is 10 days.		<i>Impact on baseline hypothesis</i> HQa: XP techniques in RUP will lower the values in HQ1, HQ2 and HQ3. HQb: Simpler components will lower the values in HQ1, HQ2 and HQ3.	

Table 6.1: GQM abstraction sheet on the introduction of XP in RUP.

**H1** XP techniques in RUP will lead to lower defect density in the finished product.

**H2** XP coding techniques will lead to less effort spent in RUP's implementation discipline.

**H3** XP techniques in RUP will lead to shorter development increments.

### 6.2.1 H1: XP techniques in RUP will lead to lower defect density in the finished product.

This goal focuses on the number of defects as a quality indicator in the finished product. A system that is perceived to be free of defects may be considered of high quality, although this is not necessarily true. For example, a defect free system that does not meet the non-functional requirements on performance is not of high quality. We will therefore assume that the system meets all functional requirements, to make the number of defects per code volume unit a valid measure of quality.

Our hypothesis is that XP techniques used in RUP lead to a lower defect density in the finished product. The defect density is the number of defects found per use case point. If our hypothesis holds, the defect density will be lower with XP and RUP than with plain vanilla RUP.

### 6.2.2 H2: XP coding techniques will lead to less effort spent in RUP's implementation discipline.

We want to find out whether introducing XP techniques into RUP will lower the effort used on development. The hypothesis is that XP's coding techniques lead to more rapid development, and thus less effort spent on implementation.

To test this hypothesis we will measure the number of person hours spent on each use case. Measures on person hours per use case has already been gathered for other projects in the company, thus providing us with baseline data.

### 6.2.3 H3: XP techniques in RUP will lead to shorter development increments.

Both RUP and XP are iterative development processes, but the former usually recommends longer iterations than does the latter. We want to see whether using elements from XP will reduce the length of the iterations in RUP. We will measure this by recording the length of each iteration in days.

Our hypothesis is that introducing XP elements will shorten the length of the iterations in RUP. Reference data to be used in testing the hypothesis are project plans from the earlier phases of the project.

## 6.3 Metrics on variation factors

From table 6.1, we can deduce metrics to validate the hypotheses given in the previous section. The resulting metrics will be presented below, linked to the Quality focuses and Variation factors from table 6.1.

Note that even though we are primarily interested in the effect of introducing XP elements in RUP, we will also take into account the properties of the development project at hand. This is necessary since several variables affect the quality of a software system. Looking at the development process in isolation may give a false image.

### Metrics for HQ1

We need two metrics to test baseline hypothesis HQ1 in table 6.1. These metrics are:

**M1** Number of defects per use case.

**M2** System volume.

The number of defects will be measured during system testing, and defects will be registered by use case. If a defect stems from more than one use case, it will be registered for each use case. We consider a failed functional test as a defect, and a single fault in the system may be the source of several defects. Code volume will be measured in use case points [ADSJ02]. We will gather lines of code as reserve metric if use case points prove to be immature for this use. The defect density ( $\theta$ ) is calculated using formula 6.1.

$$\theta = \frac{\sum M1}{M2} \quad (6.1)$$

### Metrics for HQ2

Baseline hypothesis HQ2 relies on the amount of development effort per use case. This metric will be called M3, and is a simple count of person-hours spent on each use case.

### Metrics for HQ3

Our last baseline hypothesis concerns the length of the development iterations, and the accompanying metric is the number of calendar days spent per iteration. This metric is named M4.

#### Variation factor metrics

The quadrant named *Variation factors* in table 6.1 gives the two controlled variables in the case study. The following list gives the metrics attached to the these variation factors.

**Qa** Development method.

**M5** Either plain RUP or RUP with XP

**Qb** Use case properties.

**M6** Complexity of each use case on the scale *High, Medium or Low*.

We are primarily interested in the effect of variation factor Qa, but to make the study more robust, we have also included other factors of a project that we assume can have implications on the baseline hypotheses. These extra factors are included in Qb, which is a “catch-all” category for project complexity and size. The reason for adding these metrics is to allow a finer separation of the data, to show that use cases with comparable properties fare better when developed with XP techniques in RUP.

#### 6.3.1 A summary of metrics

Table 6.2 shows a summary of the different metrics that will be gathered. The metrics have received a scale type, giving a clue on how they will be treated during analysis. The notion of scale types has been taken from [WRH<sup>+</sup>00].

Metric	Name of measure	Scale type
M1	Defects per use case	Interval
M2	Use Case volume	Interval
M3	Programming effort per Use Case	Interval
M4	Length of iterations in days	Interval
M5	Development method	Nominal
M6	Use Case complexity	Ordinal

Table 6.2: A summary of metrics.

## 6.4 Case study context

The case study was conducted on a project in a software development firm. The project developed a system for a bank, and the study concentrated on the development of a single component. Three developers worked on the component under study.

The reference data are taken from another project where similar functionality was developed. Data gathered in this project includes test-logs, use case models and development effort. From these data we have made the baseline hypotheses.

# Chapter 7

## Case study results

This chapter is divided into two main sections. The first presents the results of the case study, while the other discusses the implications of the findings. Lastly, the context of the case study is discussed.

### 7.1 Results

This section presents the results of the metrics from table 6.2.

#### 7.1.1 Metrics on Quality focus

**M1: Defects per use case** Table 7.1 sums up the total number of defects found for the case and the reference project. The table also presents the average number of defects per use case. We note that  $\theta$  and the average are much lower for the case project than for the reference project. The numbers are incomplete, as the system testing was not done at the time of writing.

	Case Project	Reference Project
Use Case Points	77	70
Number of Defects	10	34
Defect Density ( $\theta$ )	0,13	0,49
Number of Use Cases	8	11
Average # of Defects per UC	1,25	3,1

Table 7.1: Defects per use case point and use case

**M2: Use case size** The use case size has already been included in table 7.1 in the row Use Case Points. The calculation of use case points can be found in appendix A. The defect density is calculated using formula 6.1

As an alternative measure on size, table 7.2 shows the numbers of lines of code for the two projects.  $\theta$  is in this case calculated as the number of defects per thousand lines of code. With this basis for the  $\theta$  calculation, the case project has twice the defect density of the reference project.

	Production code	Test code	Total code	Defect Density ( $\theta$ )
Case Project	1487	694	2181	4,6
Reference project	13736	905	14641	2,3

Table 7.2: Lines of Code and defect density.

**M3: Programming effort per use case** Table 7.3 presents the total implementation effort for the two projects. For the case project, the effort spent on pair programming is also specified. We note that a fifth of the effort was spent on pair programming, suggesting that the technique was not entirely embraced by the developers. The table shows the average effort per use case. The case project use cases demanded on average almost half the effort compared with the reference project.

	Case Project	Reference Project
Total Development Effort	349	846,5
Pair Programming Effort	68,5	N/A
Pair Programming Percentage	20%	N/A
Number of Use Cases	8	11
Average Effort per Use Case	44	77

Table 7.3: Programming effort in hours.

**M4: Length of iterations in days** The case had three iterations set aside for development, and table 7.4 shows the length of each iteration to be exactly ten days. The iteration lengths for the reference project is given in the rightmost column of the table, and the results for the reference project are identical to the case.

Iteration	Case project	Reference project
1	10	10
2	10	10
3	10	10

Table 7.4: Length of development iterations in days.

### 7.1.2 Metrics on Variation factors

**M5: Development method** This metric has only two possible values: RUP and RUP with XP. The reference project uses RUP while the case project uses RUP with XP. The case project used the three lowest levels from the development case presented in chapter 5; unit testing and test first, pair programming and refactoring.

**M6: Use case complexity** The use case complexity has been employed in the use case point calculation presented in appendix A. It has not been used for any other purposes. The complexities are presented in the use case models in the appendix.

## 7.2 Discussion on case study results

In the following passage, we will discuss the implications of the data unearthed during the case study. Based on this discussion, we will reject the three hypotheses presented in chapter 6. The hypotheses are repeated here for quick reference:

**H1:** XP techniques in RUP will lead to lower defect density in the finished product.

**H2:** XP coding techniques will lead to less effort spent in RUP's implementation discipline.

**H3:** XP techniques in RUP will lead to shorter development increments.

When analyzing the results, we noted that the use cases were not as comparable as initially assumed. The use cases in the reference project were far bigger than those of the case project. They also differed in scope and complexity. Any results obtained by statistical methods like ANOVA or Mann-Whitney would be invalidated by these weaknesses, so this discussion will take a qualitative rather than quantitative approach.

**M1: Defects per use case** In table 7.1, we see that  $\theta$  for the case project is lower than the  $\theta$  for the reference project. The discussion on metric M2 will show why we will disregard this number, as the use case point calculation is flawed. The remainder of this passage will define  $\theta$  as the number of defects per thousand lines of code.

$$\theta = \frac{\#Defects}{KLOC} \quad (7.1)$$

Even though the number of defects found for the case project is preliminary, we note that the defect density of the case project is twice that of the reference project. We must note that during testing of the reference project, the entire system was tested from the user perspective. The same method has been used in testing the case project, but the case project is but one component of the system tested. This fact invalidates any results of statistical methods, since the projects are incomparable. The discussion on defect density is therefore qualitative rather than quantitative.

Several defects found during the case project testing are unrelated to the component developed, because of the scope of the tests. The actual number of defects found in the component developed in the case study is lower. The testers have at the time of writing found only one defect directly related to this component. With this assumption,  $\theta$  for the case project is 0,5 for the preliminary data. This is a noticeable improvement from the reference project, but since the final results are not yet ready, the numbers are not valid.

As a remark on the name of this metric and its use in the discussion; the number of defects per use case has only been used in computing the total number of defects in the system. Presenting the numbers at this level of detail has therefore been judged unnecessary.

Sadly, the team was not able to deliver the system test results on time. Thus, we cannot conclude whether the change of development method has improved quality. We will therefore reject H1: *XP techniques in RUP will lead to lower defect density in the finished product.*

**M2: Use case size** The use case size given in table 7.1 is calculated using the method and data specified in appendix A. Some assumptions on the data have been made, and this passage will present the rationale behind these assumptions.

Figures A.1 and A.2 in the appendix presents the use case models for the two projects. To compute the values, each use case in the model has received a relative complexity by the developers, as metric M6. After calculating the use case point value of the reference project with this assumption we arrived at a use case point value of 70.

Function points are often used to estimate effort. We will see if use case points can be used in the same way. To find an effort estimate based on function points, we multiply the function points by a constant. We will do the same with our use case points. Research have showed that a constant between 15 and 30 is of proper size. This constant will be adjusted according to company experience, but a good first value for the constant is 20. Table 7.5 presents the results of the effort estimates for the use case points.

Project	UCP value	Constant	Estimate	Actual	Error
Reference	70	20	1397	846,5	39%
Case	77	20	1531	349	77%

Table 7.5: Effort estimates based on the use case points.

The estimate for the reference project overshoots the actual effort with almost 40%. This is problematic, but research on use case points is still immature. Adjustments on the weight of the different technical and environmental factors will have to be made, and parts of the overestimate can be attributed to these sources. In addition, the constant used in the estimates have to be adjusted according to the company that uses it, but a single project does not give enough data for a recalculation.

Use case points for the case project are actually higher than the value for the reference project. This is a problem, since we know from interviews with the developers that the case project is the smaller of the two. We will therefore not put great faith in the effort estimate for the case project in table 7.5. Since we do not have enough data to adjust the constant, we will not use this estimate.

The problem with the use case point calculation is that it does not differentiate on the size of the use case. From the calculation in appendix A, we note that the number of use cases are more or less equal for the projects. The difference is that the reference project's use cases have a larger scope, demanding a bigger code volume for implementation. This variation is not visible in use case point calculation today, thereby making the numbers less useful for the kind of calculation used in this study.

During system testing, the use case models of the two projects were further decomposed into smaller use cases. An attempt was made to calculate the use case points with these alternative, but still representative, use cases. The motivation was to see if these use cases would provide a more accurate, and therefore more useful result. In this calculation, the reference project got a value of 150, while the case study got 135. Although these numbers reflect the fact that the case project is the smaller, the overall difference in the value based on our knowledge of the projects is still insufficient.

We can conclude that at this point, neither use case point estimation nor the amount of data are advanced enough to be useful in the analysis of the case project. Further

research in the field of use case point estimation may improve this. As long as the size of the use case is not incorporated in the calculation, it is not sufficient for our needs.

**M3: Programming effort per use case** From table 7.3, we see that the total effort for the case project is far below the value for the reference project. The average effort per use case is also half the size for the case project compared with the reference project. This may give some foundation for H2.

The contexts of the two projects are quite similar. Developers work part time on the system, and the technical challenges are similar. Both projects are concerned with the use and presentation of persistent objects in a web application. A detailed presentation of the technical and environmental factors of the projects can be found in appendix A.

We can explain the lower effort in the case project by the variation factor M5 and the project context. M5 is concerned with the development methodology on the project, and we might claim that the introduction of the development case presented in chapter 5 is the source of the improvement.

A contrasting theory is that the case project relies heavily on existing components. These components were developed in the reference project, implying that some of the effort recorded in the reference project was related to infrastructure. From this angle, we can claim that the case project has profited from the work on the reference project. This will not support our hypothesis.

	Case project	Reference Project
Effort	349	846,5
Lines of code	2181	14641
Effort / KLOC	160	57

Table 7.6: Effort per thousand lines of code (rounded).

Lastly, the size of the case project is far smaller than the reference project when we consider the code base of the implementations. Table 7.6 presents the effort per thousand lines of code. We note that the effort per thousand lines of code is almost three times as big for the case project as for the reference project. The source of this difference can be that the developers use more time to write test suites, to define interfaces and review each others code during pair programming. Inexperience with pair programming can also explain the result.

When we consider the average effort per use case, we get a favorable result in support of the hypothesis. Based on our knowledge of the size and scope of the use cases for the two projects, we know that this result is skewed. Table 7.6 shows that the case project spent a greater effort per lines of code on implementation than the reference project. This last argument weighs heavier in the final evaluation than does the first, thus we will reject H2: *XP coding techniques will lead to less effort spent in RUP's implementation discipline.*

**M4: Length of iterations in days** From table 7.4, we note that the iterations in the two projects have equal length. This implies that our modification of RUP has not improved this facet of the development process.

On the other hand, all developers working on the project were working part time on the case. We might claim that our modification has allowed the developers to stay within the given limits in spite of a difficult work context. A counter argument to this theory is that the reference project also was a part time project for the developers.

Based on this, we will reject H3: *XP techniques in RUP will lead to shorter development increments.*

**M5 and M6: Development method and Use case complexity** Development method is our basic variable, and all the data discussed in the preceding section are affected by its value. The use case complexities are used in the use case point calculation, and gives a hint on which use cases are the most difficult to implement. The implications of the complexity and the development method have been discussed in the sections where they have been used, so no further comment on the results are given here.

### 7.2.1 Threats to validity

The following paragraphs will discuss the validity of the results in the study. An explanation of these terms may be found in [WRH<sup>+</sup>00].

**Conclusion validity:** The study is qualitative. No statistical methods have been used, therefore the study does not have conclusion validity.

**Internal validity:** Internal validity is threatened by the selection of motivated developers. The team under study had at an earlier point shown interest in the techniques studied. They are therefore not representative for all developers in the company.

**Construct validity:** The case project was a part-time project, and the techniques under study are generally described as advantageous in a full-time setting. It is also threatened by mono-operation bias, in that the only variable under complete control was the development method.

**External validity:** No obvious threats to external validity exists, as the case study was an actual project in the company. It reflected the normal workday for the participants.

From the previous points, we conclude that the results from the study can only be qualitative rather than quantitative.

## 7.3 Discussion on the project context

This passage will present some statements from the developers and team leaders on the case project. They are presented here as a supplement to the preceding discussion.

### 7.3.1 Anecdotal evidence

From table 7.2, we see that the reference project has an order of magnitude more production code than the case project. The case project has a much higher percentage of test code compared with the reference project. More than a quarter of the code is test code in the case project, compared with less than a tenth for the reference project. This is not significant towards  $\theta$ , but it is an interesting point related to perceived quality.

As can be seen in the list below, the developers had an impression that the quality of the case project was higher than the reference project. This confidence in the system may stem from the amount of test code, verifying the different parts of the system for the developers.

Table 7.3 shows that a fifth of the development effort in the case project was with pair programming. This result shows that in a setting where developers work part time, synchronizing the workday is difficult. The developers were disappointed by this result. Pair programming itself received a favorable review. The developers solved difficult problems in less time, and with higher perceived quality. On simple tasks, the technique did not add much value. Therefore, the case team said they would continue to use pair programming on difficult tasks, but program solo on the simple ones.

Some arguments against this approach are in order, since assessing the complexity of a task in advance is a serious challenge. How does one know for sure whether a task is complex or not? In addition, these so-called simple tasks will not benefit from the review process inherent in pair programming, lessening the confidence in these parts of the system.

During the development iterations, and after a short period of adjustment, the developers stated the following on the XP techniques:

- They are fun to use.
- They are useful in transferring knowledge about the system.
- Test first forces the developers to define detailed interfaces at an early point.
- All parts of the system developed under XP influence were thoroughly reviewed.
- Perceived quality was higher.
- Have clearly identifiable coffee cups (we don't want to transfer our colds to each other).

The project itself was judged moderately successful by the team members. The customer of the system did not complain too much about the delay of the delivery. On the other hand, the company finally tested the programming techniques of XP. This was something they had wanted to do for a long time.

### 7.3.2 Research difficulties

During the case study, some communication problems occurred between the researcher and the case project. All the team members contributing to the case project had other

projects at the same time. The case project had a low priority for the company. This delayed the start up of the project, as well as leading to considerable delays. The system was delivered several weeks too late.

The communication between the company and the researcher was a problem. All information had to be exorted by showing up in person, as mails were not answered. No information was given before a question was asked. An example of this was when the project leader said the day before the Easter holidays that the implementation phase of the project, where the measurements relevant to the case study occurred, was to start the day after the holidays were over. This was stated on the phone after electronic mail messages had been ignored for several weeks. The reason for the delay was the low priority of the project, and that the managers did not know when the project would start.

# Chapter 8

## Case study conclusion

In this last chapter, we will present a conclusion on the results from the case study. Thereafter, we will give some suggestions for further work.

### 8.1 Conclusion

From chapter 6, we remember the following three hypotheses regarding the combination of the Rational Unified Process and Extreme Programming.

**H1:** XP techniques in RUP will lead to lower defect density in the finished product.

**H2:** XP coding techniques will lead to less effort spent in RUP's implementation discipline.

**H3:** XP techniques in RUP will lead to shorter development increments.

The results from the case study, as presented in chapter 7, does not give enough material to support H1. H2 and H3 are rejected based on the findings in the study. Thus they are all rejected for the context described in this thesis.

Qualitative evidence does suggest that the combination has certain advantages, and the team under study is willing to continue to use the combination. This is in contrast to the lack of quantitative evidence supporting the theory of this thesis.

Employees in the company where the case study took place have found the research method interesting. They liked the results of the GQM approach, and have said that they will use the metrics proposed in this thesis for further process improvement in their company.

Employing use case points in defect density calculation is novel, and its use was rather experimental. The discussion around defect density concludes that use case point estimation is too immature for this kind of work at the moment. Several flaws in the method forced the researcher to discard the data on use case points, the most important reasons being:

- The technical and environmental factors were not calibrated for projects like the case and reference project.

- The method disregards use case size in the estimation. Thus a use case model with many small use cases will get a value in the same range as a model with equally many but larger use cases.

## 8.2 Suggestions on further work

Attempting to do a case study on the same theme with full time workers is the most interesting path to follow. Conducting a multiple case study on the development case used in this case study is also possible.

Testing the ideas from chapter 4 in an actual project and on the XP community is another possible path. Both posting the ideas directly to the user community and producing a paper is possible.

The results from the study can provide input to the research on use case point estimation. Calibrating the environmental and technical factors is one alternative. Accommodating use case size is another.

An obstacle frequently encountered during the study was the lack of material on using COTS with XP. Building a framework that permits the integration of COTS in an XP project would be interesting.

# Appendix A

## Use case point calculation

This appendix will discuss the use case point calculation of the case study and the reference project. A presentation on use case point estimation based on [ADSJ02] is given before the calculation of the use case points start. Note that use case points as a basis for defect density calculation is a novel use of the technique.

Reference project	70
Case project	77

Table A.1: Use case point values for the two projects.

### A.1 Method for computing use case points

To compute the use case points of a model, we need several different values. The following passage will present the method for computing these values. The values are:

- Unadjusted Actor Weight (UAW)
- Unadjusted Use Case Weight (UUCW)
- Unadjusted Use Case Points (UUCP)
- Technical Factor (TF)
- Environmental Factor (EF)

The computation of these values is carried out as follows.

**UAW:** Actors are categorised as *simple*, *average* or *complex*. The categories are weighted as follows:

**Simple:** weight 1

**Average:** weight 2

**Complex:** weight 3

The UAW is computed by counting the number of actors in each category and multiplying each total by its weight. The UAW is the sum of these products.

**UUCW:** Use cases are in a similar way categorised as *simple*, *average* or *complex*. They are weighted as follows:

**Simple:** weight 5

**Average:** weight 10

**Complex:** weight 15

The UUCW is calculated in the same way as the UAW. Multiply the number of use cases in each category by its weight; then sum the results.

**UUCP:** The UUCP is the sum of the UAW and UUCW.

**TF:** To compute the technical factor, one must first compute the *Tfactor*. This number is calculated by multiplying each technical factor weight with its value, and summing the results for each factor. The TCF is computed using formula A.1. Table A.3 contains an example of a TFactor calculation.

$$TCF = 0,6 + (0,01 \times TFactor) \quad (A.1)$$

**EF:** The environmental factor is computed in the same way as the technical factor. Summing up the product of the weight and value of each factor and using formula A.2. Table A.3 contains an example of an EFactor calculation.

$$EF = 1,4 + (-0,03 \times EFactor) \quad (A.2)$$

**UCP:** The final use case point value is derived from the previous numbers by formula A.3

$$UCP = UUCP \times TCF \times EF \quad (A.3)$$

## A.2 Use case point calculation for the reference project

Figure A.1 presents the use case model for the reference project. Each use case has a complexity that is used in computing the UUCW. The complexities have been assigned post mortem, thereby taking into account the difficulties the developers encountered when programming.

UAW and UUCW calculation can be found in table A.2. The resulting UUCP is *96*. Table A.3 gives the Technical and Environmental Factors of the reference project. Using formula A.3, we arrive at a use case point value of *70* for the reference project.

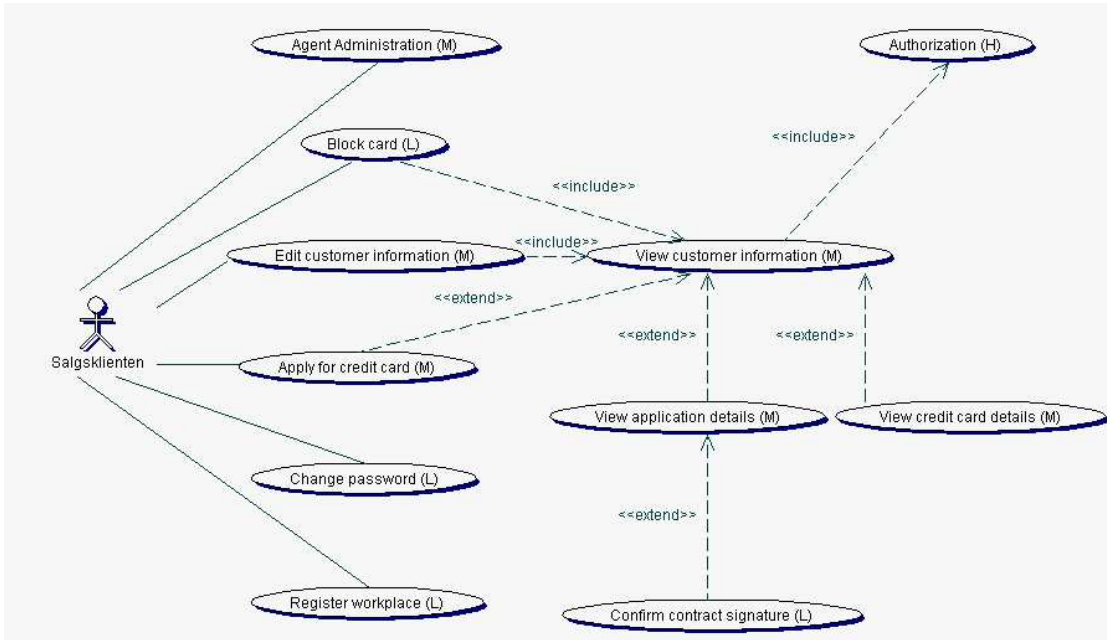


Figure A.1: Use case model of the reference system.

Complexity	Amount	Weight	Value
Simple	1	1	1
Average	0	2	0
Complex	0	3	0
UAW			1
Simple	4	5	10
Average	6	10	60
Complex	1	15	15
UUCW			95
UUCP			96

Table A.2: Unadjusted Actor Weight and Unadjusted Use Case Weight for the reference project

Factor	Weight	Value	Total
T1	2	5	10
T2	2	4	8
T3	1	0	0
T4	1	2	2
T5	1	4	4
T6	0,5	2	1
T7	0,5	2	1
T8	2	3	6
T9	1	3	3
T10	1	4	4
T11	1	5	5
T12	1	3	3
T13	1	0	0
Tfactor			47
TCF			1,07

Factor	Weight	Value	Total
F1	1,5	4	6
F2	0,5	4	2
F3	1	5	5
F4	0,5	4	2
F5	1	4	4
F6	2	3	6
F7	-1	1	-1
F8	-1	0	0
Efactor			24
EF			0,68

Table A.3: Technical and Environmental factors for the reference project.

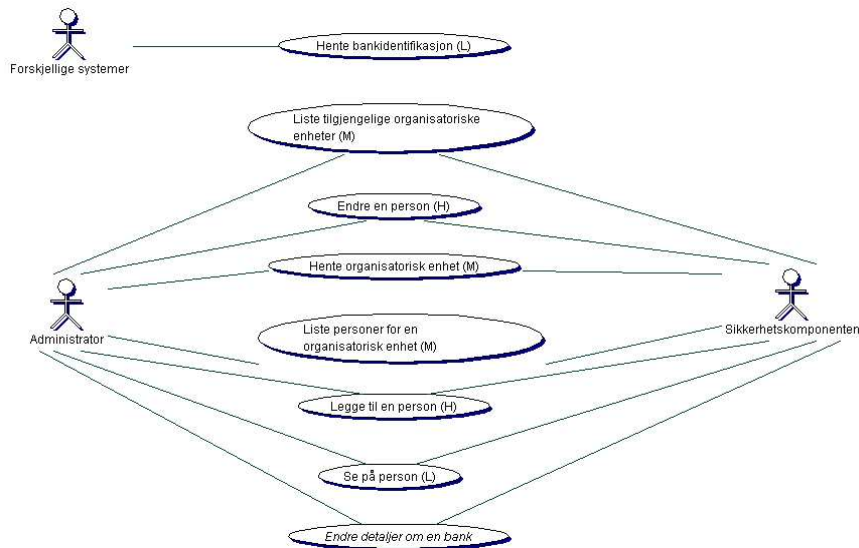


Figure A.2: Use case model of the case system.

### A.3 Use case point calculation for the case project

The use case point calculation for the case project is done in the same way as for the reference project. Figure A.2 presents the use case model.

UAW and UUCW calculation can be found in table A.4. We arrive at an UUCP value of *80*. During system testing, the model changed somewhat, and another use case of low complexity was added to the model. Thus, the value presented above did not stabilize until late in the project. This was expected, as the requirements for the system was not written in stone. Table A.5 gives the Technical and Environmental Factors for the case project. Using formula A.3, we arrive at a use case point value of *77* for the case project.

Complexity	Amount	Weight	Value
Simple	2	1	2
Average	0	2	0
Complex	1	3	3
UAW			5
Simple	3	5	15
Average	3	10	30
Complex	2	15	30
UUCW			75
UUCP			80

Table A.4: Unadjusted Actor Weight and Unadjusted Use Case Weight for the case project

Factor	Weight	Value	Total
T1	2	5	10
T2	2	4	8
T3	1	4	4
T4	1	3	3
T5	1	3	3
T6	0,5	1	0,5
T7	0,5	2	1
T8	2	3	6
T9	1	4	4
T10	1	4	4
T11	1	4	4
T12	1	0	0
T13	1	0	0
Tfactor			47,5
TCF			1,075

Factor	Weight	Value	Total
F1	1,5	3	4,5
F2	0,5	5	2,5
F3	1	5	5
F4	0,5	0	0
F5	1	4	4
F6	2	3	6
F7	-1	5	-5
F8	-1	0	0
Efactor			17
EF			0,89

Table A.5: Technical and Environment factors for the case project.

# References

- [ADSJ02] Bente Anda, Hege Dreiem, Dag I. K. Sjøberg, and Magne Jørgensen. Estimating software development effort based on use cases—experiences from industry. In Martin Gogolla and Chris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada.*, pages 487–503. Springer Verlag, 2002.
- [Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
- [BF01] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison Wesley, 2001.
- [BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language Users Guide*. Addison Wesley Longman, 1999.
- [Coc02] Alistair Cockburn. *Agile Software Development*. Addison Wesley, 2002.
- [EK01] Jutta Eckstein and Rolf F. Katzenberger. XP inside the trojan horse: Refactoring the unified software development process. In Giancarlo Succi and Michele Marchesi, editors, *Extreme Programming Examined*, pages 137–154. Addison Wesley, 2001.
- [Fow99a] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, 1999.
- [Fow99b] Martin Fowler. *UML Distilled: Applying the Standard Object Modeling Language, second edition*. Addison Wesley Longman, 1999.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [Her95] Paul Herzlich. RAD and quality principles. In *Will Tickit and ISO9000 Survive Rapid Application Development?*, pages 1–5. IEEE, 1995.
- [INC] <http://www.idi.ntnu.no/grupper/su/inco.html>.
- [JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Pub co, 1999.

- [Kru99] Phillippe Kruchten. *The Rational Unified Process, An Introduction, second edition*. Addison Wesley, 1999.
- [MS95] Don Millington and Jennifer Stapleton. Developing a RAD standard. *IEEE Software*, pages 54–55, September 1995.
- [Pee01] Vera Peeters. Simple design and unit testing with Enterprise JavaBeans: The box metaphor. In *2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*. XP2001, May 2001.
- [Pol01a] Gary Pollice. RUP and XP, part I: Finding common ground. The Rational Edge, <http://www.rational.com/products/rup/links.jsp>, March 2001.
- [Pol01b] Gary Pollice. RUP and XP, part II: Valuing differences. The Rational Edge, <http://www.rational.com/products/rup/links.jsp>, April 2001.
- [Smi01] John Smith. A comparison of RUP and XP. <http://www.rational.com/media/whitepapers/TP167.pdf>, 2001.
- [WKCJ00] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17:19–25, jul-aug 2000.
- [WRH<sup>+</sup>00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: An introduction*. Kluwer Academic Publishers, 2000.
- [Yin94] Robert K. Yin. *Case Study Research: Design and Methods, second edition*. Sage Publications, 1994.