

Making the RUP Agile

S. Salin

Abstract

The Rational Unified Process (RUP) is a software engineering process framework developed by Rational Software Corporation. Many organizations have rejected the idea of using the RUP because the first thing they notice about it is that it contains a huge amount of process documentation and guidance. This study explores some of the ways that enable the RUP to be customized and made agile. One way to customize the RUP is by building a new RUP Process Configuration, i.e. the base RUP is modified with plug-ins that either bring new data to the base RUP or modify the existing elements in the base RUP. There are various plug-ins available but plug-ins can be also developed from scratch. In addition to the best practices that the RUP promotes, there are other key principles that aim at making the RUP lightweight – these are presented briefly in this paper. Probably one of the easiest solutions is to use one of the predefined configurations that the RUP offers, for example, RUP for Small Projects. This configuration can then be further redefined if necessary. One way to make the RUP agile is to combine it with some agile methodology such as XP. This means that certain XP practice(s) such as the test-first design is incorporated with the RUP. Yet another way to make the RUP agile is to use the RUP Plug-In for XP, available from Rational Software. This plug-in contains two configurations: the XP_with_RUP and the RUP_with_XP. The first one adds RUP practices for small projects and the second one adds XP practices to the RUP. Results found of applying the RUP in an agile manner are few. This paper presents experiences of a company named Zuhlke Engineering AG from using the RUP in an agile manner in two projects. Finally, this paper presents a process, which is a combination of the RUP and XP – it is called dX.

1. INTRODUCTION

In this era of strong enthusiasm towards agile software development processes the interest of making also the Rational Unified Process (RUP) agile has increased. The RUP has a reputation of being a heavy process and a lot of convincing and training needs to be done before the RUP gets rid of this reputation. A common misunderstanding is that the RUP is a ready process to be taken into use and partly this is true, however, the RUP is actually a process framework that should not be used as such but should always be modified to the project's or organization's specific needs. The RUP can be adapted for small, medium, and large projects. This, however, requires some effort and usually, especially if the organization is new to the RUP, there will be an experienced mentor involved. What comes to agility of the RUP, it can be made just as lean as needed, there are basically no limits. In fact, making the RUP agile often means customizing it to a point where everything considered unnecessary for the project in question has been eliminated. Special attention should be given to the quality aspects in the project if the goal is to make

the process as lean as possible by eliminating everything that seems extra. Whether agility is achieved, for example, by leaving out most of the documentation or by minimizing testing – both are decisions that will have an effect on the quality.

1.1 Background

The RUP contains a huge amount of documentation, guidance, and details of how to develop quality software. This is one of the main reasons why the RUP has not been tempting for many organizations. Usually after once browsing the RUP it is not understood that what you see is not necessarily what you will get. There is a clear challenge in convincing projects to use the RUP and to assure people that the RUP can be made as small and compact as needed. A great challenge also lies in making the RUP agile – convincing that it can be made agile is not enough, actions speak for themselves. The RUP can be misused in various ways and the core idea of the RUP, iterative development, can be missed in many dangerous ways, eventually turning the RUP into a waterfall model. It is important to recognize the pitfalls of thinking when starting to adopt the RUP. It is also important to realize that configuring the RUP is not a one-time effort but a continuous process of improvement and customization.

The purpose of this study is to help in recognizing the pitfalls of the RUP thinking and to show that yes, the RUP can be tailored for almost all kinds of software projects.

1.2 Research Problem

This study seeks answers to the following questions:

1. Can the RUP be made agile and if, then *how* can it be made agile?
2. Are there results reported from practical use of the RUP in an agile manner?

While searching answers for these questions, the quality concerns in the agile RUP are taken into consideration.

1.3 Objectives of the Research

The main objectives of this study are:

1. To show that it is possible to make the RUP as agile as wanted and/or needed by customizing the RUP to fit the specific needs of each project. This goal in mind, this study presents various possibilities of building and applying an agile version of the RUP.

2. To present practical experiences of use of agile RUP, for example, project reports and an example process based on the RUP.

In addition to these main objectives, this study evaluates how software quality can be maintained while making the RUP agile.

1.4 Scope

This study has the following limitations:

- This study is not a tutorial for the RUP or any other process or methodology.
- Practical evidence shown is based on only one report of the use of agile RUP. At the time of writing this study no other reliable reports were found from the sources used for this study.

1.5 Methodology

This study was conducted mainly based on material found in literature and articles. The RUP version 2003.06.01 and the RUP Builder version 2003.06.01 provided by Rational Software Corporation were also used as a source of information and for hands-on experiments.

1.6 Structure of the Report

The study begins with presenting the RUP and explaining its main elements and concepts. Discussion of ways how the RUP is generally misused ends chapter 2. Chapter 3 presents various ways how to customize the RUP and how to make it agile. A report of making and applying the RUP in an agile manner in a company named Zuhlke Engineering AG is presented. Also included in chapter 3 is a discussion of how to produce quality products with agile RUP and a minimal process implementation of the RUP, called the dX, is presented. Finally, a brief introduction of how to choose the right amount of process and which RUP configuration to use is presented.

2. RATIONAL UNIFIED PROCESS (RUP)

“RUP provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end users, within a predictable schedule and budget. RUP captures many of the best practices in modern software development in a form that can be tailored for a wide range of projects and organizations.” (Rational Software, 2003)

2.1 Definition of the RUP

The Rational Unified Process developed by the Rational Software has many kinds of definitions in the literature. The

reason for this may be that RUP is still unknown to many and that RUP can be used in so many ways that definitions do not do justice to all the RUP’s possibilities. To the question “What is the RUP”, Kroll and Kruchten (2003, p. 3) provide the following answers:

- The RUP is a software development approach that is iterative, architecture-centric, and use-case-driven.
- The RUP is a well-defined and well-structured software engineering process that clearly defines who is responsible for what, how things are done, and when to do them.
- The RUP is also a process product that offers a customizable process framework for software engineering.

The RUP is pre-populated with a large amount of process know-how captured over the last fifteen years by Rational personnel. The RUP framework is an open framework, which is updated twice a year.

2.2 RUP’s Main Elements and Concepts

In order to be able to work with the RUP some familiarity of the concepts is needed. Here is a brief introduction to the concepts that are discussed later on in this paper.

The RUP has two structures: dynamic structure and static structure. In figure 1, vertical dimension represents the static aspect of the RUP and horizontal dimension represents the dynamic aspect of the RUP.

The dynamic structure deals with the lifecycle dimension of a project, dividing the project into four *phases*: Inception, Elaboration, Construction, and Transition. Each phase contains one or more iterations – there are as many iterations as needed to achieve the business objectives of that phase, but no more. (Kroll & Kruchten 2003, p. 10-12)

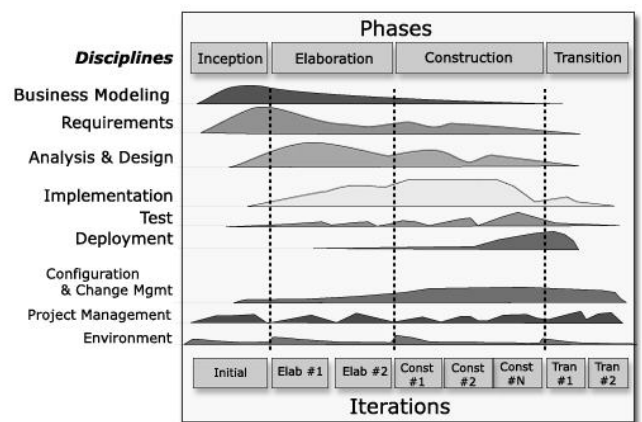


Figure 1 Overall Architecture of the RUP (Rational Software Corporation 2003)

The phases and their content in brief, are (Kroll & Kruchten 2003, p. 10-12):

Inception. High-level understanding of all the requirements and the scope of the system are established. The business case for building the system is produced and the decision whether to proceed with the project, is done.

Elaboration. An executable architecture is designed, implemented, and tested. Implementing and validating actual code address major technical risks such as resource contention risks, performance risks, and data security risks.

Construction. Most of the implementation is done. Several releases are deployed to ensure that the system is usable and meets the user needs. Deploying a fully functional beta version of the system ends this phase; it must be noted that the system will still probably require tuning of functionality, performance, and overall quality.

Transition. The product is tested for release and minor changes may be done based on user feedback. No major changes are done, only fine-tuning of the product.

The static structure deals with how process elements – activities, disciplines, artifacts, and roles are grouped into the process disciplines. (Kroll & Kruchten 2003, p. 13) The key elements of the RUP are (Kroll & Kruchten 2003, p. 13-18):

Roles. One individual usually has one or more roles. In the RUP the roles define how the individuals should do the work and what kind of competence and responsibility that individual should have.

Activities. A unit of work that an individual is asked to perform in a specific role. An activity usually takes a few hours to a few days to complete and usually involves one person. Activities may be repeated several times on the same artifacts when going from one iteration to another – by the same role, but not necessarily by the same individual.

Artifacts. Tangible project elements, a piece of information produced, modified, or used by a process. Artifacts are both outputs of activities and inputs used by roles to perform other activities. Examples of artifacts include: use-case model, design model, class, Vision, Business Case, source code, and executables.

Workflows. Workflows describe sequences of activities that produce some valuable result and they also show interactions between roles. Disciplines are the most common workflows. A workflow can be expressed as a sequence diagram, a collaboration diagram, or an activity diagram.

Disciplines. Disciplines are logical containers that group together all process elements – roles, activities, artifacts, and additional concepts such as guidelines and templates. There are nine disciplines in the standard RUP: Business modeling, Requirements management, Analysis and design, Implementation, Deployment, Test, Project management, Change management, Environment.

The RUP promotes many proven software development

practices and the ones that are given high visibility in literature are (Kruchten 2002, p. 6-14):

Develop iteratively. Iterative approach to developing software means that each successive iteration builds on the work of the previous iterations to evolve and refine the system until the final product is complete. Each iteration has well-defined objectives and includes most of the development disciplines.

Model visually. A model being a simplification of reality it helps visually to understand and shape both the problem and its solution. This is important especially in the case of a complex system. Visual modeling, when done with a standard modeling language such as the UML (Unified Modeling Language) helps in communicating the decisions and changes to the whole team.

Manage requirements. Identifying a system's real requirements is a continuous process and requirements are expected to change throughout the development lifecycle. Managing requirements contains at least the following tasks: eliciting, organizing, and documenting requirements, evaluating changes, assessing their impact; and tracking and documenting trade-offs and decisions.

Control changes. Coordinating the work results of many developers involves establishing repeatable workflows for managing changes to software and other development artifacts. Coordinating iterations and releases involves establishing and releasing a tested baseline at the end of each iteration.

Continuously verify quality. Quality is the responsibility of each member of the development organization. The RUP describes what models are needed, why they are needed, and how they are constructed.

Use component-based architectures. Applications built with components are more resilient to change. Components also facilitate reuse allowing building higher quality applications fast.

2.3 Common Mistakes with the RUP

There are factors that inhibit the successful adoption of the RUP and usually lead to undesired results. According to Larman et al. (2001) there are recognizable patterns in these failures. They describe seven this kind of patterns, or mistakes. The first mistake described is *superimposing the waterfall thinking*. The most common misinterpretation, conscious or unconscious, of the RUP phases is to consider them similar to the waterfall phases, i.e.:

- Inception – do most of the requirements.
- Elaboration – do the detailed design and models.
- Construction – implement.
- Transition – integration, system test and deployment.

This type of thinking suggests that first all the requirements are defined and stabilized. Then the design is done and fixed,

after which implementation is done based on the design. Finally the process ends in integration, system testing, and deployment. Waterfall thinking has turned out to be the first, best, and most common strategy for total RUP failure. Some things can be built like buildings but software has not turned out to be one of them. (Larman et al. 2001)

Applying the RUP as a heavy, predictive process. A “heavy process” is a pejorative term connected to such qualities as for example rigidity and control, lots of documents, significant process overhead on top of the essential work, and predictive rather than adaptive. Predictive means for example, that the entire iterative project is planned in detail, most or all of the RUP artifacts are created, and a lots project and process formal ceremony is added. (Larman et al. 2001)

Avoiding object technology skills. To really fail with the RUP, having or educating people with deep object skills is ignored. If the project doesn’t have skilled object technology developers, no amount of process is going to save that project. (Larman et al. 2001)

Undervaluing adaptive iterative development. If an organization is moving to the RUP, and does not experience a deep transformation at many levels in how they think about developing software, this is probably a sign that they didn’t truly adopt iterative development. Properly understood, iterative development is like a revolution – especially if an organization is moving from waterfall values and practices. (Larman et al. 2001)

Avoiding mentors who understand iterative development. If an organization is starting its first iterative RUP project, the worst mistake it can do is to include inexperienced people on the project – people that have never done short adaptive iterative development before. (Larman et al. 2001)

Adopting the RUP in a big bang. Introducing the RUP to the entire organization in a short period of time will certainly face resistance. To make sure that adopting the RUP fails, people are not trained, people are not explained why the RUP is adopted, there are no small pilot projects to start with, and there is no mentor. (Larman et al. 2001)

Taking advice from misinformed sources – if everything that is read or heard about the RUP is accepted without any critical thinking, the chances of failing with the RUP are increased. Even among the so-called experts, there are those who don’t really understand iterative development and the information they present is not correct.

3. MAKING THE RUP AGILE

The RUP has received a lot of critics about being too heavy and document-driven, requiring too many artifacts to be produced. As Fowler (2003) points out, the RUP is trying to be everything and in the end it offers nothing. Kruchten (2002) on the other hand sees the RUP as a rich palette of knowledge from which to choose what is needed.

Regardless of all doubts it is possible to make the RUP very lightweight, agile. The purpose is not to make the RUP

another XP but to heavily scale down the out-of-the-box RUP when necessary. The RUP is a process framework and is not meant to be used as such but it should always be tailored according to the needs of the project and/or organization at hand (Kruchten 2002).

The best practices that many agile approaches such as XP are based on have been around for many years. These practices include iterative development, continuous integration and primary focus on the executable software, refactoring, coding standards, and user stories. However, they have also introduced best practices of their own such as pair programming and test-first development. (Kroll & Kruchten 2003, p. 52)

Making the RUP agile is not very difficult. The RUP’s existing basic approaches such as iterative development already offer a good starting-point for this. The RUP can be made agile in various ways; the most obvious being that the RUP is customized to be as small as possible to meet the specific needs of the project. Another, seemingly popular way is to combine the RUP with some agile method, such as XP.

3.1 Customizing the RUP

Using the RUP out-of-the-box is possible but the overwhelming amount of documentation and the high level of detail makes many professionals turn their back on the idea of adopting the RUP, especially for small teams and fast paced projects (Hirsch 2002). But as Kruchten (2000, p. 31) states, no process should be followed blindly. Instead, the process must be made as lean as possible while not forgetting its mission to rapidly produce predictably high-quality software. Being a process framework, the RUP enables organizations to modify, adjust, and expand it to accommodate their specific needs, characteristic, and constraints (Kruchten 2000, p. 31).

The RUP framework is made up of *RUP Base* (also known as *base RUP*) and *RUP Plug-Ins*. The RUP Base contains the elementary parts like general definitions (such as the definition of the key best practices), concepts, and principles. More process components can be added to this base RUP in the form of the RUP Plug-Ins. RUP Plug-Ins are precompiled RUP Process Components that can either bring new data to the base RUP or they can redefine and specialize elements that already exist in the base RUP. (Kruchten 2002) As a result, a new RUP Configuration can be produced. It must be noted that plug-ins can be also built by the organizations themselves if they want to do further RUP customization than the IBM Software and its partners are offering (Kroll & Kruchten 2003, p. 189).

Configuring the RUP for a project consists of two steps (Kroll & Kruchten 2003, p. 179):

1. Producing the *RUP Process Configuration*, i.e. deciding what parts of the RUP to use.
2. Producing personalized views, called *Process Views*, into the RUP Process Configuration.

To produce the above, a tool named *RUP Builder* within the RUP product is used. The RUP Builder allows to make the process either smaller or larger and of higher ceremony or lower ceremony by enabling the linkage of RUP Plug-Ins into the base RUP. RUP Builder also allows defining how formally the work is to be done, e.g. whether to use more comprehensive document templates or lighter templates. (Kroll & Kruchten 2003, p. 180-181).

After deciding which plug-ins are included in the configuration and which process components within the plug-ins and RUP Base are used, RUP Builder validates the package and publishes a RUP Web site from the new configuration. (Kroll & Kruchten 2003, p.181)

The Process Views are personalized views into the RUP Configuration, i.e. depending on the role and responsibility, different content is shown. Process Views are also created in RUP Builder. (Kroll & Kruchten 2003, p.182)

It is not enough, however, to build new RUP Process Configurations. Each project is unique and therefore it must be decided how to apply the existing RUP Process Configuration for each particular project. A *development case* is an artifact that provides specific process guidance for a project or group of similar projects. E.g. it can guide which artifacts the project should produce, which tools or templates to use, when to produce them, and with what level of formality. The development case does not instruct on *how* to produce an artifact, instead it links to the relevant RUP Configuration, which in turn provides the *how* part. The development case can be very brief, about four to eight pages long, and therefore produced quickly. (Kroll & Kruchten 2003, p.184-185)

Building the configuration is not hard – that is technically done in a few minutes. The hardest part is to understand what is available and what to select to the configuration – this may take time. (Kroll & Kruchten 2003, p.182)

3.2 Agile RUP

IBM Rational has taken agile software development seriously and it has produced many documents on how to make RUP agile. In one of these papers, written by Kruchten (2002), it is stated that an agile process is not simply about the size of the process or the speed of delivery; it is mainly about flexibility. Or, as Evans (2003) puts it, making the RUP agile is not a violation of the RUP philosophy; making RUP agile simply requires that people make their own decisions about what they will use and do on their projects.

3.2.1 Key principles in the agile RUP

In addition to the best practices (listed in chapter 2.2) that the RUP has strongly promoted, there are also other key principles that the RUP is based on but which are less visible and easily forgotten (Kruchten 2002):

- Develop only what is necessary.
- Focus on valuable results, not on how the results are

achieved.

- Minimize the production of paperwork.
- Be flexible.
- Learn from your mistakes.
- Revisit your risks regularly.
- Establish objective, measurable criteria for progress.
- Automate what is human intensive, tedious, and error prone.
- Use small, empowered teams.
- Have a plan.

Many of these key principles clearly aim at making the RUP more lightweight.

Iterative development, which is one of the six best practices in the RUP, has the greatest impact on process agility. Developing software incrementally also enables to fine-tune the process itself; after each iteration project participants learn more and based on what has worked and what has not they can adjust the development process, guidance, activities, and artifacts. (Kruchten 2002)

3.2.2 Increasing Process Agility

According to Kruchten (2002), the RUP framework achieves process agility through one of its disciplines that covers the *process engineering* aspects. In practice this means that the RUP framework contains everything (for example, activities, artifacts, guidance, and tools) needed to evolve the framework, i.e. to create your own RUP configuration. Kruchten (2002) presents four ways that can further improve the RUP's process agility and help organizations to get started with the RUP philosophy:

Predefined configurations. Instead of providing the out-of-the-box RUP, there should be a selection of predefined configurations that already have some choices made for certain type of software development environment. An example of the predefined configuration is *RUP for Small Projects*. In this configuration, elements that are not likely to be used in a small, “five-people-for-five-weeks” projects have been eliminated.

Componentized RUP. This allows the customization of the RUP that is discussed in chapter 4.1. By adding plug-ins to the base RUP the organization can tailor the RUP for their specific needs.

Tool support for RUP configuration. A tool named Rational Process Workbench supports building of additional plug-ins. This tool is a process-modeling tool that uses UML (Unified Modeling Language) to model the process.

Tools for process authoring. By making available those very tools that Rational used to develop the RUP, Rational is opening up a process marketplace. Organizations can replace parts of the RUP with their own ideas.

3.2.3 Combining the best of XP with the RUP

Another interesting view on how to make RUP agile is to

combine the RUP with some full-blooded agile methodology. The idea is to incorporate some practices from the agile approach into a RUP-based project, however, it must be noted that including some XP practices in RUP will not make RUP identical to XP. For example, Smith (2001) and Pollice (2003) have made comparisons of XP and RUP – how they differ, overlap, and what XP practices are suitable to be used in RUP, and what are not. Examples of XP practices that Smith (2001) finds beneficial to be also employed in RUP are:

Pair programming. RUP does not describe code production at a fine-grained level so this can definitely be used in a RUP based process.

Test-first design and refactoring. These fine techniques could certainly be applied in the Implementation discipline of the RUP.

On-site customer. Many of RUP's activities would benefit greatly in terms of increased efficiency through having a customer on-site. With the customer as a team member, the need for many intermediate deliverables, especially documents, can be reduced.

Pollice (2003) recognizes several consistencies between RUP and XP. According to him, RUP and XP are not necessarily exclusive, although they arise from different philosophies. By incorporating techniques from both methods, one can arrive at a process that helps to deliver better quality software and quicker than before (Pollice 2003). One particular practice from XP that Pollice (2003) recommends to be adapted to RUP is the testing practice. In XP, before any code is written a test for that code is written first. Another great testing principle in XP is that customers write acceptance tests to ensure that the system does what it is supposed to do. Adding these practices to complement the already existing RUP testing practices gives an excellent quality focus to the project Pollice (2003).

3.2.4 RUP Plug-In for XP

As a last technique presented of how to make the RUP agile, is the *RUP Plug-In for XP* that Rational Software first introduced in September 2002. This plug-in provides guidance for the teams who are using RUP and are looking for effective ways to apply it to small teams and projects. Version 2.0 of the RUP Plug-In for XP is contains two predefined RUP configurations. The *XP_with_RUP* configuration describes XP, and adds RUP practices for small projects. The *RUP_with_XP* configuration describes RUP for a medium-sized project, and adds XP practices that may be useful. The RUP Plug-In for XP offers specialized guidance for developers by combining the strengths of the core elements of the Rational Unified Process - including iterative development and the utilization of component-based architectures - with XP's agile approach. The RUP Plug-In for XP offers small and medium-sized teams an out-of-the-box set of practices, roadmaps and activities that help these teams to focus on their ultimate goal: generating high-quality, executable code.

(Pollice & Martin 2002)

3.3 Agile RUP in Practice

Reports of applying RUP in an agile manner are already available. One such report written by Michael Hirsch (2002) describes in detail how the RUP was made agile, how it was applied and what were the experiences of applying the agile RUP on two small (teams of three to five developers) projects, referred here as Project A and Project B.

Customization of the RUP on these projects was done in the following areas (Hirsch 2002):

- Artifacts. Radical scale down of the artifacts, only the ones that were really needed and added value were kept.
- Activities. Activities were not used for detailed iteration planning. Activities were treated like a textbook: if a developer needed guidance on how to complete an assigned task, he or she would consult the appropriate activity descriptions to find answers.
- Roles. No roles were assigned; those were used only to verify that all the required skills were present.
- Project planning. As the RUP suggests, the projects kept two levels of plans – one on a high level and the other one in detailed level. The detailed plan was made for each iteration and was prepared a few days before the start of an iteration.
- The RUP phases were adopted without modifications.
- Iterations. Iterations were about four weeks long, however the length of an iteration had nothing to do with how frequently the builds were made and integrated to the system; daily builds were made a rule in all the RUP projects.
- Project control. Weekly status meetings were used where each developer estimated the remaining work to achieve his or her iteration goals. If necessary, an iteration was ended with fewer features on time rather than with all the features but too late.
- The RUP online documentation was not changed because the same modification work would have been necessary to repeat with every new release of the RUP.

These were the guidelines in both projects. There were some differences between the projects such as the following:

- On the Project A, a customer representative was present in all iteration planning and iteration assessment meetings. On Project B, customer involvement was minimal.
- Project A decided to skip formal testing, i.e. no test plan or test cases were prepared, instead, ad-hoc testing became the practice. Project B did unit testing with JUnit, they wrote a test plan and use cases.

It turned out that Project A was a success but Process B was a total failure. The reason? The main reason was that on Project B, the customer was extremely passive, e.g. during the first four months of the project, no feedback was received from the customer. What Hirsch (2002) sees here as a key lesson is that no software process, regardless how sophisticated, can compensate for customer feedback. Hirsch (2002) otherwise believes that the RUP configuration they used was in the true spirit of agile methods. He reports that after these two projects they have completed about a dozen more projects with the agile RUP.

Based on experience, Hirsch (2002) lists the things to consider when making RUP agile:

- Carefully select a small subset of artifacts.
- Iteration planning should primarily focus on results (as opposed to focus on tasks to be done).
- An iteration is a planning period, not a single build.
- Iterations of one month work well.
- Even on small projects, insist on a person from the customer with at least 50% of his or her time available for the project

Project A was the first official RUP project at Zuhlke Engineering AG. During this project no formal testing was done. This is definitely against the basic RUP philosophy. Testing in the RUP is considered extremely important and is considered to be the responsibility of each member in the team. As Hirsch (2002) states in his article, that even though there were no quality problems, the team never felt secure about the system's quality. Later on, the company has added unit testing with one of the xxUnit frameworks, a test plan, and documented test cases for system testing.

3.4 *Quality in Agile RUP*

One of the principles of the RUP approach is "Make quality a way of life, not an afterthought". This means that ensuring high quality is not only a responsibility of the testing team but that it is the responsibility of every team member in all parts of the lifecycle. (Kroll & Kruchten 2003, p. 6)

Quality is good, but not quality at any cost. The RUP approach to testing incorporates the concept of Good Enough Quality (GEQ) from the work of James Bach (1997), which in a nutshell means that quality is ultimately situational and subjective; the goal is to understand the problems and benefits of a situation well enough to be able to eliminate the right problems and deliver the right benefits. Most organizations practice some sort of "good enough reasoning" about their products - the only ones that don't are those that lack the imagination and skill to see how their products might be improved (Szymkowiak & Kruchten 2003).

When making the RUP agile, there are some issues that clearly have an affect (either positive or negative) on the

quality of the project deliverables. This study has found the following to be the most common:

Iterative development. Iterative development allows initiating testing early, already in the Inception phase. Enabling early feedback may result in significant time and cost savings. Iterative development also forces to test more often and enables to constantly verify the quality of the deliverables. In general, the RUP requires development and testing to be parallel activities, which means that e.g. artifacts are reviewed as they are produced, identifying requirements also includes considering how they can be tested, and testing should be closely coupled with design. Most projects adopting the RUP report that the first tangible result of the improved process is an increase in quality. (Kroll & Kruchten 2003, p. 47)

Testing. Iterative development is also the heart of agile approaches and therefore continuous testing is also part of them. According to Szymkowiak and Kruchten (2003), the RUP approach to testing has become more attuned to iterative development, less focused on high ceremony, and closer to harmonizing with the XP mantra of "test first". Making the RUP agile does not mean that testing should be reduced to a minimum. According to Smith (2001), XP techniques that can be applied in the RUP's Implementation discipline are the test-first design and refactoring - especially the test-first design is an excellent way to clarify requirements at a detailed level. The practice of not writing test documentation or doing any systematic testing is not an acceptable way to make the RUP agile. This approach was used in one of the projects that Hirsch (2002) reported. Although Hirsch (2002) reported no serious quality problems, he admitted that it was not a good approach.

Documentation. Another questionable way to make the RUP agile is to leave out most of the documentation and let the code be the document. According to Pollice (2003), having only the code as documentation is not enough to make it possible to maintain the system, especially in the case of object-oriented ones. The code is at too low a level of abstraction to really tell what the system, as a whole, is trying to do Pollice (2003). Some time spent capturing and maintaining design documents reduces the risk of misunderstanding and can speed up the development Pollice (2003).

No matter which practices are chosen to make the RUP agile, no such short cuts should be taken that endanger making of quality products. Making the RUP agile requires some work and familiarity with the framework, selecting the right elements and just enough of process is not easy and that is why many organizations rely on the ready-made RUP configurations, such as the one for small projects.

3.5 dX – a Minimal Implementation of RUP

It is justified to say that dX, a minimal RUP process, is a combination of RUP and XP. It is built on the same phases that the RUP uses: Inception, Elaboration, Construction, and Transition but many of the practices deployed in these phases are adopted from XP. The phases and their contents in dX are (Martin 2001):

dX Inception. During this phase the customer writes simple descriptions of major use cases on index cards. The customer representative must be part of the team at all times. Based on the use cases, simple prototypes are built. The purpose of the prototypes is to measure how fast the developers are, whether the use cases have the right size and detail, and what are the potential system architectures. The code in this phase may be later discarded.

dX Elaboration. The customer continues writing the use case cards. The developers estimate the amount of work for each use case card. The customer then prioritizes the use cases based on how big a risk they pose. Iterations are planned in this phase; the length of an iteration should not exceed one week. Each iteration is analyzed and designed to fit into the system architecture identified during Inception. It is up to the developers whether they choose to use UML diagrams or not, those are not required.

The design is then committed to code. In dX every line of code produced must be examined by two pairs of eyes. Testing is of high priority in dX. In fact, unit test code is written before the actual code that it will test. The tests and the code grow together.

Designs and code are kept as simple as possible; no complexity is added unless mandated by a use case. In order to produce quality code and to ensure that it fits into the system architecture, developers are encouraged to change any code that does not meet these requirements. The tests that are continuously run reveal any problems that may appear after changes. All team members jointly own code, which means that anyone can modify any part of the code. Integration is done at least daily.

dX Construction. The Construction and Elaboration phases are almost indistinguishable. The only difference is in the stability of the architecture and the project plan.

dX Transition. Although the system is still very under-functioned, it will be live from now on. Iterations will continue until new release can be made alive.

dX relies heavily on communication between the team members; therefore an open workspace is required where team members and customer representatives can be in close contact with each other. Overtime is considered a failure of the process and only two consecutive weeks of overtime is allowed. If this is not possible, then the project plan is changed.

Martin (2001) also emphasizes that even though it may be necessary to produce some intermediate artifacts to support the software, they are not the goal for the process. Only the

software is the goal and this should always be kept in mind when constructing a process from the RUP framework.

3.6 Which Way to Choose

Kroll & Kruchten (2003, p. 58) show the different RUP Configurations on a process map, figure 2. Low Ceremony means that the process produces minimum supporting documentation and has little formalism in the working procedure. High Ceremony on the other hand means comprehensive supporting documentation and traceability maintained among artifacts, and so forth. On the vertical axis Waterfall means a linear approach with late integration and testing. Iterative approach is risk-driven with early implementation of an architecture and early integration and testing. (Kroll & Kruchten 2003, p. 50)

Making the RUP agile is not always desirable. When it comes to complex systems, contractual and regulatory requirements, large teams, and distributed teams, the high-ceremony approaches are justified. These approaches often produce quality systems that are easy to maintain, but the production costs are higher, and the time-to-market is slower than for low-ceremony approaches. If the time-to-market is a primary concern, high-ceremony approaches may actually produce lower quality simply because there is not enough time to follow all the process details. As a result, before doing any customization of the RUP product, projects should first decide where they want to place themselves on the process map. (Kroll & Kruchten 2003, p. 53, 64-65)

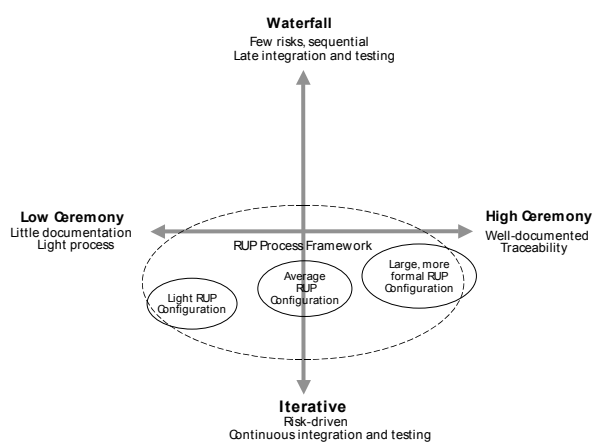


Figure 2 RUP Configurations on the Process Map

4. SUMMARY AND CONCLUSIONS

One of the main goals of this study was to find out whether the RUP be made agile. The result of this goal is that yes, it is possible to make the RUP agile. But in the case of the RUP, making it agile is mostly about customizing it in various ways and that is what this study ended up investigating. As Hirsch (2002) puts it: being agile is a mindset that can be practiced with many processes, including lightweight RUP. Some software process approaches are more refined to be agile, they offer well-defined practices and rules that are straightforward and easy to follow, like XP's pair programming – there is a lot of guidance on how to apply that in practice. This kind of process can be used almost without any customization. The RUP is another story. The RUP should never be used as it comes out-of-the-box, especially not the Classic Configuration as it contains just about every document and guideline one can imagine belonging to a software development process. Even the smaller configurations are not actually small in size. All configurations of the RUP can be further reduced by eliminating everything that is not considered to bring any value. This is not only about being agile; it is about using common sense. There are many ways to make the RUP agile. One way is simply to customize the RUP itself. Another way, which is often presented in literature and articles, is to incorporate some of the XP practices with the RUP. The RUP Plug-In is yet another, more concrete way to introduce XP practices to the RUP.

Offering various ways to customize the RUP is a brilliant thing but it doesn't really eliminate the work that has to be done when the RUP is adopted; there is no one-size-fit-all process package available. When the RUP is customized, only after the first iterations it can be evaluated whether the process should be further refined and this evaluation probably continues till the end of the project. Customization gets easier after more experience and knowledge about the RUP's possibilities. It is a good idea to involve an experienced RUP mentor in the customization process. This helps avoiding the common pitfalls, such as implementing the RUP in a waterfall way.

Another main goal of this study was to find evidence and results of practical use of the agile RUP. Reports from using agile RUP were very few compared to the material found on how to make the RUP agile. One such report used in this study is written by Hirsch (2002). His report describes not only successful projects using agile RUP but also failures and lessons learned from these failures. It also provides concrete instructions, based on experience, on how to make the RUP agile. Another practical example of agile RUP is provided by Martin (2001) who's presentation of a minimal RUP process named dX is presented in this study. Martin (2001) states that the dX process has been used on several successful projects – however, he does not give any information about these projects. Martin's article seems more like a description of XP

disguised in the phases of the RUP. As a result, this source cannot be considered as a true example of the use of agile RUP.

One goal of this study was also to look into the quality aspects while making the RUP agile. This goal in mind, some quality concerns rose from Hirsch's (2002) report. For example, agility of the RUP had been partly achieved by questionable way - leaving out testing. Ignoring totally something important like testing or documentation from the process while making it agile soon turns the process against its original idea; the process may be agile but instead of helping to develop high-quality software, the process helps to develop software that may be behaving unpredictably, is difficult to maintain, and does not satisfy the customer.

Another thing that may be forgotten in the process of chasing agility is that not every kind of software development project is amenable to agility. For example, if the software to be developed is very complex, large and critical, or the team developing the software is big and geographically distributed, a more traditional process approach is suitable.

Finally, to further study the agile RUP more concrete actions should be taken – to get experience from customizing the RUP and using it in an agile manner in real projects would give more practical insight into the research effort.

REFERENCES

- Bach, J. Good enough Quality: Beyond the buzzword. IEEE Computer, 1997, vol. 30, no 8. pp. 96-98.
- Evans, G. 2003. Agile RUP for non-object-oriented projects. The Rational Edge September 2003. http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/sep03/m_rupagility_ge.pdf
- Fowler, M. 2003. The New Methodology. <http://martinfowler.com/articles/newMethodology.html>
- Hirsch, M. 2002. Making RUP Agile. OOPSLA 2002 Practitioners Report. ACM Press.
- Kroll, P., Kruchten P. 2003. The Rational Unified Process Made Easy: a Practitioner's Guide to the RUP. Addison-Wesley. 416 p. ISBN 0321166094.
- Kruchten, P. 2000. The Rational Unified Process: An Introduction. 2nd edition. ISBN: 0201707101. Addison-Wesley Longman Publishing Co., Inc.

- Kruchten, P. 2002. Agility with the RUP. The Rational Edge January 2002.
<http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/archives/jan02.html>
- Larman, C.; Kruchten, P.; Bittner, K. 2001. How to Fail with the Rational Unified Process: Seven Steps to Pain and Suffering.
http://www.agilealliance.org/articles/articles/How_to_Fail_with_the_RUP_-_Kruchten_and_Larman.pdf
- Martin, R.C. 2001. The Process.
<http://www.objectmentor.com/resources/articles/RUPvsXP.pdf>
- Pollice, G. 2003. Using the RUP for small projects: Expanding upon Extreme Programming. <<http://www-106.ibm.com/developerworks/rational/library/409.html>>
- Pollice, G.; Martin, R.C. 2002. The Rational Unified Process and Extreme Programming: An Introduction to the RUP Plug-in for XP. TP 150, 12/02. Rational Software White Paper. Rational Software Corporation.
http://www.rational.net/content/images/catpulse/catapulsemember/attachment/pdf/7500_2131_final_tp150_rup_xp.pdf (Registration needed)
- Rational Software Corporation. 2003. Rational Unified Process Version 2003.06.01.06.
- Smith, J. 2001. A Comparison of RUP and XP. Rational Software White Paper. <<http://www.rational.net>>
- Szymkowiak, P; Kruchten, P. 2003. Testing: The RUP Philosophy. The Rational Edge February 2003. <http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/feb03/RUPphilosophy_TheRationalEdge_Feb2003.pdf>