Proceedings of the 5th WSEAS International Conference on Telecommunications and Informatics, Istanbul, Turkey, May 27-29, 2006 (pp98-104)

Permeation of RUP and XP on Small and Middle-Size Projects

KREŠIMIR FERTALJ, NIKICA HLUPIĆ, DAMIR KALPIĆ
Department of applied computing
University of Zagreb Faculty of Electrical Engineering and Computing
Unska 3, Zagreb 10000, CROATIA

Abstract: Modern software development business, as a very dynamic and often risky process, imposes new approaches to planning and organisation of the work. At present, a typical team can apply one of the agile methodologies, among which RUP and XP are the most common. They are both well-established and proven in practice, but nowadays it is clear that they cannot respond to all the new challenges separately. The gap between them leaves uncovered exactly those issues that mostly affect small and middle-size projects, which are the majority of all projects in modern business. This paper proposes an integral process, a combination of RUP and XP, which should be more convenient for small and middle-size projects than RUP or XP alone. Generally, it anticipates less documentation than RUP and suggests more planning than XP, trying to adopt the best form of both and adjust them to the modern business. Properly applied, the proposed process should be more acceptable and more efficient than other similar methods.

Keywords: Software development, Methodology, Process

1 Introduction

The development of Extreme Programming (XP) in nineties [1, 2] initiated lively discussion in software development community between those promoting XP and those advocating its "older big brother" Rational Unified Process (RUP). A number of years they were almost confronted methodologies for software development, and there were numerous studies [1-3], which tried to prove general superiority of one over another. However, recently (in the last few years), we have become more and more aware of their complementarity [8-10], and of need for integration and permeation of their concepts. This is a mere consequence of the fact that most of the present software projects fit into the class of middle-sized projects, meaning that XP is too "little and firm", and RUP too "big and universal" [1] for literal and strict implementation. Releasing and adjusting some of the directives, and combining ideas of both of these philosophies, is a logical solution for filling the gap between them and a natural evolution of software development methodologies in the present business world.

2 Short overview of the RUP and XP

2.1 Overview of the RUP

Rational Unified Process, originally Unified Development Process [11] is a software development framework intended as the process complement to Unified Modelling Language (UML) [11]. As a universal software development framework, RUP can accommodate wide variety of processes. Because of wide scope and generality, it represents highly systematic and quite disciplined approach to the software development. Although it provides a number of predefined "templates" sometimes called out-of-the-box roadmaps [9], which model different types of processes suited for different types of software development projects, it does not require any specific activity or production of any specific artefact, nor does it require Rational tools for effective application. It simply provides guidelines to help the user to "tailor" the framework and decide what is needed and applicable to a specific task. This enables the user to choose the subset of artefacts that will be produced and even to create its own artefacts, if there are no appropriate ones in the predefined set.
RUP emphasises the adoption of certain rules, the so-called "best practices" [9, 7] of modern software development. These practices are already proven by experience in many projects and in various teams, and as such believed to be desirable and effective way to reduce the risk inherent to the software development projects. The best practices are [9]: Iterative development, Management of requirements, Application of component-based architectures, Visual modelling, Continuous quality verification, Control and tracking of changes.
The iterative development is one of the most important practices in RUP. It provides constant feedback, serving as a kind of project self-control and as the main mechanism for reduction of inherent risks.
The RUP lifecycle comprises four basic phases, as follows:

I. Inception

During Inception, all parties in the project must agree about the main aims (scope) of the project, the predictable time schedule and the basic architecture of the system. The main products of this phase should be a Vision document, initial Use-case model and Preliminary Project Plan. The Vision document is a key artefact produced in this phase. It is a high-level description of the system, which describes what the system is, to whom it is intended, what features it must provide to the customer and what constraints exist.

II. Elaboration

The goal of the Elaboration phase is to clarify the design of the system architecture into more detail in order to make its implementation in Construction phase easier and more straightforward. The key for success is identification of the most important requirements, which have the largest overall impact on the system, and assessment of risks. With identified requirements and risks, the architecture can evolve from the basic, outlined in Inspection phase, to the more detailed, which can provide a stable basis for Construction phase. Elaboration phase usually has two iterations. The aforementioned activities should be accomplished in the first iteration, and in the second one we plan the activities for the following phases. The main new documents that must be produced in Elaboration phase are Software Requirements Specification, Software Architecture Document and a plan of further activities in the forthcoming phases.

III. Construction

Although in the beginning of the Construction phase there is still a lot of work on the design of the system, the Construction phase is a manufacturing process, where the developers have to create executable system. The system is improved through iterations and should provide more and more features. It can also undergo significant modifications in response to possible changes of specifications or use-cases. In the late iterations, the focus of development effort gradually passes from the overall functionality to development and testing of particular system components. In addition, late iterations are the right time for the team to create an initial plan for performing the acceptance tests, produce training materials and sketch the Deployment Plan.

IV. Transition

Transition phase focuses on the availability of the software for its end users. This is the time for the first release and for a test of the fully functional system, usually called beta-test. The system has to be tested by the customer in regard to all aspects of the intended usage. User feedback should help the fine-tuning the product, but not rarely, users tend to significantly revise and change their requirements after the first usage of the whole system, so in the Transition phase there must be a special concern about Change Management.

2.2 Overview of Extreme Programming

Extreme programming is one of the methodologies that have attracted the most attention. XP was developed by Kent Beck in 1996 for the C3 Chrysler payroll project [2]. Originally, XP was founded on four core values, which were based on fifteen basic principles and realised applying twelve practices. However, new deliberations and recent experience led to significant revision of XP [4, 5], which is nowadays founded on five core values, based on fourteen principles, thirteen primary practices and eleven corollary practices. Here we provide just a brief overview.

The five values are: Communication, Simplicity, Feedback, Courage and Respect.

I. Communication

For XP, continual communication between the customer and development team, as well as inside the team itself, is the key for success. According to XP, such communication is realised having the on-site customer and frequent small releases of the system. The on-site customer helps the developers through the user stories and small releases provide prompt feedback about the current system.

II. Simplicity

XP insists on simplicity in every stage of the product development. Both the overall architecture and particular software components should be as simple as possible, fulfilling only the specified requirements without redundancy in functionality due to anticipated future needs. The main guarantee of simplicity should be continual refactoring of the code [6]. Moreover, XP suggests production of only the necessary documents and non-code artefacts. The code is considered the best and almost sufficient documentation by itself.

III. Feedback

XP emphasises continual testing and many short releases in order to provide reliable feedback and risk reduction mechanism. Moreover, testing is in XP the foundation of development and every programmer is supposed to write tests as they write the code or even before writing the code. This enables highly stable platform for every advance in the project and should reduce the inherent risk.

IV. Courage

Courage means ability to make and realise all needed decisions, which can help or improve project development. What is necessary must be done, no matter how hard or unpopular it is. Such kind of courage implies honesty of all team members, who must be honest to themselves and aware of their capabilities, as well as brave enough to be honest about that with the rest of the team.

V. Respect

If a project team adopted the previous four values, then this fifth one is merely the natural behaviour of such team. If the team members do not care about each other, or about the customer, and do not respect other members or their work, no methodology can help the project. Mutual respect among all interested parties in the project is necessary precondition for success.

These five values establish the basic rules, but they say nothing about how to accomplish all the tasks. Concrete directions are practices, but to be able to carry them out, the team must adopt and obey some principles in everyday work. According to Kent Beck, there are fourteen XP principles [4].

Kent Beck recognises primary and corollary XP practices [4], although some authors do some further classifications [5]. For the sake of brevity, we shall provide just the basic terms for all twenty four practices, with occasional notes.

Primary practices: Stories (User Stories), Weekly Cycle, Quarterly Cycle and Slack, Sit Together, Whole Team, Informative Workspace, Energised Work (formerly *Sustainable Pace),* Pair programming, Incremental Design (comprises two former – *Refactoring* and *Simple Design*), Test-First Programming (*Continuous Testing)*, Ten-Minute Build, and Continuous Integration. The first four primary practices have evolved from a single Planning Game practice in the first edition [2], and together present one of the most distinctive characteristics of XP, compared to other methodologies. They are often called the “embrace change” property of XP.

Corollary practices: Real Customer Involvement (formerly *On-Site Customer)*, Incremental Deployment. Negotiated Scope Contract, Pay-Per-Use, Team Continuity, Shrinking Teams, Root-Cause Analysis, Code and Tests, Shared Code (formerly *Collective Code Ownership*), Single Code Base, and Daily Deployment.

3 Permeation of RUP and XP

At first glance, it seems that RUP and XP are irreconcilably opposed methodologies. On the other hand, the reality is simple and undoubted; the business environment changes in time, and it is more and more clear that neither the RUP, nor XP alone can respond appropriately to the new software business requirements. The majority of software world is too dynamic and unpredictable for “huge and time-consuming” RUP, and software projects are too expensive and important to be left to “ad-hoc” planning in XP. It is obvious that the future demands integration and permeation of these two concepts, and here is how we see this process.

Extreme programming focuses on the code. Typical projects last a few months (up to a year), and typical teams have just a few people (up to, let us say, ten) who are always available to each other and intensively communicate. There is almost no documentation, because everything changes daily and production of documentation just slows down the development. Finally, there is the on-site customer to clarify all ambiguities, so the developers always have somebody to tell them what the program should do and what is the next highest priority. The on-site customer, pair programming and short (daily) releases are the main risk reduction mechanisms.

RUP is the opposite outmost. It is a configurable process framework, which can be adjusted and tailored according to a specific project. There are no limits to the project size, price or the team size and deployment, and the main risk reduction mechanisms are iterations in each phase and detailed documentation.

These two extremes can be best combined in areas where one of these methodologies is not appropriate and the other one is. For example, XP *Pair-Programming* practice is not always desirable in teams of only a few people, because it reduces productivity per person. Similarly, *Sit Together* is neither always possible, nor is nowadays necessary, since modern communications allow effective work and cooperation from distant locations. The same holds for the on-site customer, who does not really need to be “on-site” in order to be available and useful to the developers. On the other hand, RUP’s configurability is mostly too general for small or medium-size projects. Excessive generality ends being nothing [1] and of no use, so it has to be limited. In addition, the required documentation is too extensive to be acceptable for small teams, so it has to be reduced as well.

Inability of RUP and XP to separately fulfil expectations of modern small and medium-size projects has been noticed for some time by a number of experts, and there are analyses and endeavours to combine them. However, we still consider this an open question, because all attempts of integration have ended more or less as being an absorption of XP into the RUP [9, 10,12]. The main representative of such solutions is dX, which can be considered “a minimal RUP” [8]. dX is simplified “user-friendly” RUP, which does not insist on modelling of the system using UML diagrams, but declares only use-cases and index-cards as obligatory documentation. It also adopts several, mainly coding-related, XP practices (e.g. *Shared Code*, *Code and Tests*, *Incremental Design* and *Pair programming* etc.). Nevertheless, all this cannot change the fact that it is still predominantly RUP.

Second thing that may legitimise further deliberations and search for an alternative is insufficient or inappropriate treatment of the human factor in both RUP and XP. On one side, RUP is high-level design-oriented methodology in which people are just resources as any other, easily replaceable and individually unimportant parts of a large organisation. RUP demands detailed planning so that any low-skilled person can do the job properly, regardless of possibly poor understanding of the end purpose or functionality of the entire system. However, detailed planning demands plenty of time (slow & expensive!) and predictive future, which presupposes very precise definition of the requirements and no change of them, especially not in the late stages of the development. This is directly opposite to the software development reality in which customers ordinarily change their requirements after the first release of the complete system. Moreover, this is in conflict with the human nature and intelligence, which presses us to learn and understand the purpose of our work, and to strive to something better than we already have. This means that capable people might not be satisfied by position RUP assigned to them, what can cause loss of potentially precious individuals and long-term damage. In contrast, XP relies too heavily on people and their individual skills and knowledge. Too often, it counts on extraordinary developers who can and wish more, faster and better than the majority of people can, or are ready to. This can be true and we might have such a capable team, but complete relying on humans is highly risky because inability of only a single person to do the job as expected can endanger the whole project. Moreover, highly skilled people are usually expensive and they like challenge, which might not appear on small projects, thus might cause their dissatisfaction as well.

The solution this paper suggests is an integral process, which would evenly combine RUP and XP concepts, based on previous conclusions. This process would retain evolutionary design, RUP alike four phases and iterative nature, but it would also take much more from XP than dX does. Although it requires less documentation than RUP, it suggests more planning than XP, especially at the beginning when it encourages clear definitions and rough design of the system, possibly using UML or some other tools. In the later stages of the development it implements majority of XP's practices in due course. We shall clarify this further explaining each phase one by one. The tasks anticipated in each phase are specified in tables. Tables also separately provide the needed activities, foreseen by RUP and XP, for accomplishment of particular task, and the activities printed in bold font are supposed to be accepted in the integral process. The permeation of RUP and XP is obvious and comprehensive.

I. Preliminary study

The Preliminary study (called Inception in RUP) phase (Table 1) is the beginning of the project and as such demands seriousness and comprehensiveness. Excessive planning and design like in RUP is not desirable, as well as no planning at all as in XP, but the most important requirements and aims should be defined, clarified and written somewhere. Many projects fail just because of bad foundation made at the beginning, thus every effort made in this phase can be expected to return multiply in the latter phases. The process begins by creation of the *Vision* document, which comprises: business motivation, required system features, preconditions and constraints, risks, main use cases, initial architecture design and project schedule.

Table 1: Preliminary study phase.

Tasks	RUP	eXtreme Programming
Analysis of the requirements and business modeling	**Vision document** **Use-Case analysis**	User Stories **Communication Feedback** On-site customer
Analysis & Design	**Preliminary architecture design**	System Metaphor
Implementation	**Creation of use-cases prototypes**	
Testing	Creation of test plans	
Configuration & Change Management	**Change Control Strategy**	
Project Management	**Project Schedule**	Story Estimates

After the completion of Vision document, which is a general project description, the next step is a more precise definition of main use-cases. Initial versions of use-cases should be created by the customer, serving as the basis for more detailed analysis and discussion among the developers and customer representative. This should yield new use-case documents, containing clarified and completed descriptions (possibly supported by UML diagrams) and text documents created using RUP forms as templates. Based on these descriptions, we create main use-case prototypes, which will provide us with a valuable indication of possible shortages of the initial system architecture and enable us more precise estimation of the time needed for each iteration. Thus, the prototypes serve as the final check of the requirements and the first feasibility test. For small and middle-size projects, we do not recommend creation of test plans already in the Inception phase. Rather we suggest emphasise on *Configuration & Change Management*, that is, definition of the form and contents of *Change Request* documents, which will record every demand for change of system requirements.

At the end of Inception phase we should have clear descriptions of the main use-cases, initial system architecture (Component & Deployment diagrams) and the approximate project schedule. Vision document and a few UML diagrams supported by several other documents should be sufficient documentation for a long time.

II. System Analysis and Design

System Analysis and Design (SAD) is a counterpart of Elaboration phase in RUP and is one that combines RUP and XP in many aspects (Table 2). The main task in this phase is analysis and design of the system components, and their implementation. The analysis and design rely primarily on the RUP, and realisation on XP. Thus, they combine evenly again. In collaboration with customer, development team continues defining less important or unfinished use-cases, while those completely defined and most important get priorities and time estimates. Predictable risks are considered as well, and use-cases with high risk get the higher priority. We also set the time estimates for each iteration, which have to conform to the sum of periods foreseen for development of each use-case scheduled for a particular iteration.

The developers intensively work on analysis and design of the system. Here RUP finds its role, because XP does not define any specific activity for this purpose. We suggest that developers create models of system components and describe them by UML *Class*, *Sequence* and *Collaboration* diagrams. Class diagrams explain static structure of the system, defining the main classes, their roles and relations and, in the later iterations of analysis and design, their properties, methods and events. Sequence and Collaboration diagrams describe system dynamics, and are sometimes called *interaction diagrams*. If the need occurs for special description of a part of business process, a critical algorithm or data flow, we add an UML *Activity* diagram.

XP practices are preferable during code development. It especially holds for creating *unit-tests* before the actual development of a certain component. Creating unit-test before the components ensures that everything and in any time will do what it is supposed to do. In addition, unit-tests can be a very effective way for verification of the design of program interfaces implemented in the most important classes in the system. If it turns difficult to write coherent tests based on the accepted component interface, it is usually the first sign that something is wrong with the design. XP also anticipates *pair-programming*, but we do not consider it as obligatory. It seems reasonable to have more than one programmer on a component, but while one writes the code, the other one can write the tests to speed up the whole process. They should change their roles periodically to reduce probability of mistakes and to ensure even progression of both the code and tests. There can also be more than two programmers simultaneously working on more than one component. What is important is to retain simplicity of the system, by continual synchronisation of the code and system architecture, courage to change any part of the code and daily integration of the finished code.

Table 2: System Analysis and Design phase.

Tasks	RUP	eXtreme Programming
Analysis of the requirements and business modeling	**Use-Case analysis**	User Stories **Communication Feedback** On-site customer
Analysis & Design	**Class, Sequence, Collaboration and Activity modeling**	**Simple Design,** System design sketches (CRC sketches)
Implementation	**Architecture prototype**	**Frequent Small Releases** **Continual Integration** **Collective Ownership** **Refactoring** Pair programming
Testing	Planning, design and implementation of tests	**Test-First Programming**
Configuration & Change Management	**Change Request documents**	
Project Management	**Defined Project Plan** **Status Assessment document**	Iteration Plan

III. System Construction

The progress from the System Analysis and Design to System Construction phase is not an abrupt one, rather a gradual shift of emphasis and intensity of work from the design to implementation. Therefore, all the activities specified in Table 2 continue, as shown in Table 3.

The main difference between SAD and the Construction phase is an improved stability of the system architecture and more control over requests for changes. The change of key concepts and design in this phase of the project would require radical decisions and could seriously threaten already well-progressed work. Thus, Request Change documents created in SAD phase now become very useful and are the main protection of the system from uncontrolled or unjustified modifications of the key definitions. They are also helpful for estimation of the risk implied by the acceptance of a certain change and useful for assessment of the status of the project as well. Generally, the required documentation in System Construction phase is mostly the same as in SAD phase, with the difference that up to now everything has been sufficiently clarified and in full motion, so there is no more need for detailed planning of the future iterations as before. By the end of System Construction phase, we can start to consider the transfer (installation, testing, and education) of the system to the customer, as suggested by RUP, though it is the role of the Transition phase.

Table 3: System Construction phase.

Tasks	RUP	eXtreme Programming
Analysis of the requirements and business modeling	**Use-Case analysis**	User Stories **Communication Feedback** On-site customer
Analysis & Design	**Class, Sequence, Collaboration and Activity modeling**	**Simple Design,** System design sketches (CRC sketches)
Implementation		**Frequent Small Releases** **Continual Integration** **Collective Ownership** **Refactoring** Pair programming
Testing	Planning, design and implementation of tests	**Unit Testing**
Configuration & Change Management	**Change Request documents**	
Project Management	**Status Assessment document**	Iteration Plan

Table 4: Transition phase.

Tasks	RUP	eXtreme Programming
Analysis of the requirements and business modeling	**Use-Case analysis**	User Stories **Communication Feedback** On-site customer
Analysis & Design	Class, Sequence, Collaboration and **Activity modeling**	**Simple Design,** System design sketches (CRC sketches)
Implementation		Frequent Small Releases **Continual Integration** **Collective Ownership** **Refactoring** Pair programming
Deployment	**Deployment plan** **User documentation** **Support plan**	
Configuration & Change Management	**Change Request documents**	
Project Management	**Status Assessment document**	Iteration Plan

IV. Transition

Although formally the last, the Transition phase starts by release of the first executable version of the software, regardless of its much reduced functionality. The day of the first release is considered the system "birthday", so that the Transition phase starts very early in the project lifecycle, somewhere in the Preliminary Study phase, and continues in parallel (in "background") with SAD and System Construction phases. During Transition, customer tests the functionality of the whole system, and some minor changes and interface "polishing" are still acceptable. This is again an iterative process recognised by both the RUP and XP, but RUP provides Product Acceptance Plan document, which formalises test methods, test time schedule and criteria for successful completion of the tests, so here we recommend RUP artefacts. Simultaneously with Acceptance tests, the care should be taken about system deployment. Again, XP does not define appropriate formal procedures, so we suggest to use the RUP Deployment Plan document. Deployment plan document determines responsibilities in the team, time schedule and infrastructure prerequisites for successful system deployment.

Finally, supporting materials (user manuals, educational courses, etc.) have to be prepared, as well as the long-term maintenance plan.

4 Conclusion

RUP is a process framework, which can be applied to wide variety of projects. It is highly formal and structured, providing many out-of-the-box roadmaps for a number of project types. On the other hand, RUP does not say anything about how to actually do everything that has to be done, thus we can consider it as a completely process-oriented methodology. In contrast, XP is devoted to everyday life and low-level management of the development team. XP does not insist on documentation and does not provide any project templates. It is completely people oriented methodology, relying on human intelligence, communication and positive work-atmosphere in the team as the main guaranties of success.

Obviously, there is a gap between these two approaches, exactly where middle-size projects fit, and in this paper we presented one possible combination of RUP and XP, which should be more convenient for small and middle-size projects than RUP or XP alone. We have retained the four RUP alike phases in project lifecycle, but we have significantly reduced the documentation, selecting just those artefacts, from all of the foreseen by RUP, that are necessary to support a little larger and less compact team than expected in XP. These documents are, first of all, use-case definitions, analysis and design documentation, system architecture definition, change request documents and a few more, specified in tables 1 though 4. Everything else, besides documentation and project structure, comes from XP. We adopt reduced XP's people-orientation and most of the XP practices, especially communication, frequent small releases, code refactoring and testing, etc.; anything not precisely defined by RUP. XP practice of writing tests before or at least in parallel with code proved to be an excellent risk reduction mechanism and it is widely accepted nowadays, even in RUP processes. As it is particularly suited for small and middle-size projects, we strongly recommend it there.

The combination of RUP and XP illustrated in tables 1 through 4 is certainly possible on small and middle-size projects, and we believe that it exploits the human experience in software development more efficiently than other similar methodologies.

References:

[1] M. Fowler: The New Methodology, 2005, www.martinfowler.com/articles/newMethodology.html

[2] K. Beck: Extreme Programming Explained, Addison-Wesley 2000

[3] K. Beck, M. Fowler: Planning Extreme Programming, Addison-Wesley 2001

[4] K. Beck, C. Andres: Extreme Programming Explained – Embrace Change, Addison-Wesley 2005

[5] M. Marchesi: The New XP, www.agilexp.org/downloads/TheNewXP.pdf

[6] M. Fowler et al.: "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 1999

[7] P. Kruchten: The Rational Unified Process, Addison-Wesley 2004

[8] G. Booch, R.C. Martin, J. Newkirk: Object Oriented Analysis and Design with Applications, Addison-Wesley Longman Inc., 1998

[9] G. Pollice: "Using RUP for small projects: Expanding upon Extreme Programming", www-106.ibm.com/developerworks/rational/library/409.html

[10] G. Pollice, R.C. Martin: "The Rational Unified Process and Extreme Programming: An Introduction to the RUP Plug-In for XP"

[11] I. Jacobson, J. Rumbaugh, G. Booch: The Unified Software Development Process, Addison-Wesley 1999

[12] M. Fowler: Is Design Dead?, 2004, www.martinfowler.com/articles/designDead.html