A RUP-Based Software Process Supporting Progressive Implementation

Tiago Massoni, Augusto Sampaio and Paulo Borba[1]
Cln-UFPE, Av. Professor Luis Freire, s/n Cidade Universitária, Brazil
Tel: +55 81 3271 8430, Fax: +55 81 3271 8438, {tlm, acas, phmb}@cin.ufpe.br

ABSTRACT

In this paper we extend the Rational Unified Process (RUP) with a method that supports the progressive, and separate, implementation of three different aspects: persistence, distribution and concurrence control. This complements RUP with an specific implementation method and helps to tame the complexity of applications that are persistent, distributed and concurrent. By gradually and separately implementing, testing and validating such applications, we obtain two major benefits: the impact caused by requirements changes during development is reduced; testing and debugging are simplified.

INTRODUCTION

Software development has become a more complex activity over the last years. Clients have been increasingly demanding higher productivity, better software quality and shorter time to market. Additional strain results from new common requirements such as distribution and concurrent access. These and other non-functional aspects complicate implementation, test and maintenance activities. In order to simplify those activities, we argue that it is useful to tackle functional and non-functional concerns separately. In fact, whereas architectural and design activities should jointly consider both concerns [19], implementation activities can benefit from this separation.

An implementation method might help programmers to effectively achieve this separation. Therefore, we have defined the progressive implementation method (Pim) [6], supporting a progressive approach for object-oriented implementation in Java [7], where persistence, distribution and concurrence control aspects are not initially considered in the implementation activities, but are gradually introduced. In this way we can significantly reduce the impact caused by requirements changes during development, and tame complexity by implementing and testing different aspects of code gradually. This progressive approach is possible because this method relies on the use of design patterns that provide a certain degree of modularity and separation of concerns [14], in such a way that the different aspects can be implemented separately.

In this paper we extend the Rational Unified Process (RUP) [12] with Pim, providing proper implementation guidelines for RUP and hoping to support the progressive implementation of different aspects in software development projects where disciplined requirements, design and test activities are essential, demanding a software process. In Sections 2 and 3, we respectively present the main concepts of RUP and Pim, useful for a better understanding of our solution. Section 4 outlines the definition of RUPim, the proposed extension of RUP, and presents some results obtained in simple practical experiments using RUPim. Finally, Section 5 presents our conclusions and related work.

RATIONAL UNIFIED PROCESS

The Rational Unified Process (RUP) is an industrial software process, based on the work of Booch, Jacobson and Rumbaugh at defining the Unified Process [9]. RUP is focused on visual modeling (UML [5]) and three key ideas. First, RUP is use-case driven—use cases, which represent functional requirements in UML, drive the whole development process (planning, design, implementation, tests and deployment). Second, RUP suggests the early definition of an stable architecture supporting key use cases (including code for an architectural prototype), followed by the development of the other use cases of the application, filling the architecture baseline defined earlier. Third, RUP has an iterative and incremental life cycle, and each life cycle iteration develops a set of use cases, representing an increment to the final product [12]. This process can be described in two dimensions, as presented in Figure 1:

- The horizontal axis represents the dynamic part of the process as it is enacted, and it is expressed in terms of four serial phases, which are broken down into iterations. In the Inception phase the life cycle objectives are defined, whereas in the Elaboration and Construction phases the architecture baseline and the whole system are developed, respectively. In the Transition phase, the product is released to the user. Each phase has an iteration workflow, which suggests how to perform a typical iteration of each phase;
- The vertical axis represents the static part of the process, as it is described in terms of activities, artifacts, workers (roles played by people to perform the activities) and workflows. Workflows are set of activities conceptually related, but having also workflow details, which are small sets of activities usually performed as a single one. According to the life cycle phases, we choose which activities are performed in each iteration.

Later we discuss how these two dimensions are affected by our extension of RUP.

PROGRESSIVE IMPLEMENTATION METHOD

The Progressive Implementation Method (Pim) guides the implementation of complex object-oriented applications in Java. Using this

Figure 1: RUP iterative life cycle [16]

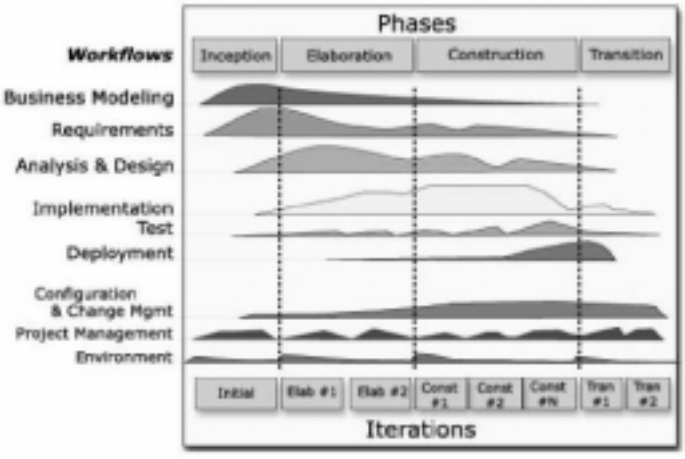
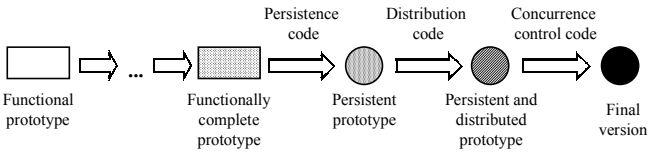


Figure 2: Progressive implementation method



method, we do not consider persistence, distribution and concurrence control aspects initially in the implementation activities. Instead, we first build functional prototypes that evolve to a functionally complete prototype. Then, the non-functional aspects are introduced, separately. Figure 2 illustrates this progressive approach.

Although the figure suggests an order for implementing each non-functional aspect, this order is not enforced by the method. In fact, the method only requires the different aspects to be implemented separately. In principle, one aspect could be implemented at the same time as another one, since they are supported by a modular software architecture.

By initially abstracting from the non-functional code, developers can, for example, quickly develop and test local, sequential and non-persistent prototypes useful for capturing and validating user requirements. As functional requirements become well understood and stable, those prototypes are used to derive a functionally complete prototype. In this way we can reduce the impact caused by requirements change during development, since most changes will likely occur before the functionally complete prototype is transformed into the final version of the application. Furthermore, the progressive approach naturally helps to tackle the complexity inherent to persistent and distributed applications, by allowing the gradual testing of the various intermediate versions of the application [6].

In order to support this progressive approach, separation of concerns principles must be applied to design activities. The software architecture must support the modular addressing of functional and non-functional aspects during coding activities. For the non-functional aspects considered here (persistence, distribution and concurrence control), this can be achieved with architectural and design patterns [1, 13], imposing some constraints on design activities. It could also be achieved by using an aspect-oriented programming language [10]. For instance, we could keep persistence-related code as an aspect, separately from the business code, and Aspect-J [11] would include persistence to the business code.

As Pim is just an implementation method, it should be carefully integrated to design and tests activities of a software process in order to be used in practice. Therefore, we used Pim in the implementation activities of RUP, integrated to its analysis, design and testing activities. The extended software process resulting from this integration is presented in the next section.

RUPIM

In order to extend RUP with Pim, defining RUPim, we have added some specific guidelines to RUP (as an example, analysis and design activities were adapted to conform to the use of specific design patterns). Similarly, we have matched corresponding concepts of RUP and Pim (activities, tasks, steps, etc.). Furthermore, we have also included new types of iterations and new activities, in order to enforce the progressive implementation of different aspects. In fact, we modified both the dynamic (phases and iterations) and static (workflows, workflow details and activities) parts of RUP.

Modifications to the Dynamic Part of RUP

As RUP promotes an iterative development, the inclusion of Pim activities to RUP workflows is insufficient for the integration. In fact, some concepts from RUP's phases and iterations were changed, in order to unify Pim's and RUP's life cycles properly. These modifications affect the entire scheduling of iterations in a development project, having impact on management activities.

The elaboration and construction phases received the major modifications, whereas the inception phase was not significantly modified, receiving some small changes related to project planning. The transition phase was not modified, since it has no direct impact on our integration. Thus, RUPim presents some alternatives that illustrate the iteration planning for the elaboration and construction phases. For these two main phases, we defined two different types of iteration: *functional iterations* and *special iterations*.

In functional iterations, the scheduled use cases must be completely analyzed and designed (as in RUP), but partially implemented. In the implementation activities, only business and user interface code is considered, and data access code is implemented using volatile data structures (Java collections, for example). In addition, distribution and concurrence are not considered, since the application will be executed in a single machine. During the elaboration phase, the architectural prototype built in functional iterations is a subset of the functionally complete prototype, as functional iterations of the construction phase complete it.

On the other hand, special iterations basically have only implementation and test activities. These activities deal with the implementation of non-functional code. Special iterations are also driven by use cases, and for each use case, persistence, distribution and concurrence control code is implemented separately. These iterations must be executed not only in the construction phase, but also in the elaboration phase, since the implementation of the non-functional code involves the most important technical risks for the architecture.

From the project manager's perspective, one or more special iterations can be scheduled for each non-functional code. However, general special iterations can be defined, addressing all non-functional code in a parallel way. Although the latter approach can optimize the productivity of the development team, in such scenario it is not always possible to isolate defects from different non-functional code, increasing complexity.

When scheduling special iterations in the elaboration and construction phases the project manager has basically two alternatives. He can schedule the special iterations as the last iterations of the corresponding phases, as showed in the example life cycle represented in Figure 3(a). So use cases will be partially implemented in functional iterations until a functional prototype is finished. Then, this prototype will evolve to a persistent and distributed application, with concurrence control, along the special iterations. Using this approach, the impact caused by changes in functional requirements during development is reduced, since they will likely not affect the non-functional code, which is implemented later in the process.

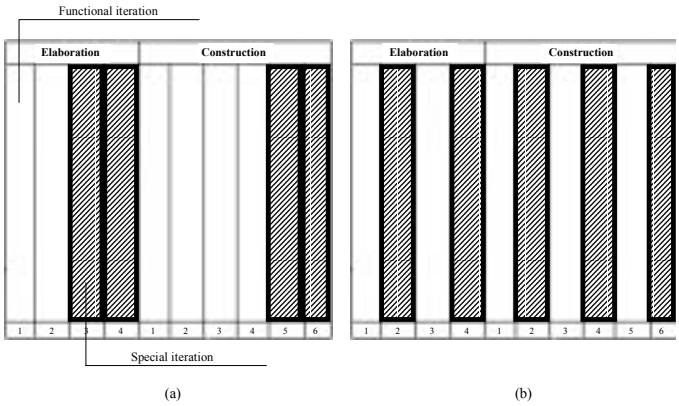
Another alternative is to plan interchanged functional and special iterations during the elaboration and construction phases, as showed in Figure 3(b). Unlike the first alternative, use cases are completely implemented, in their corresponding functional and special iterations. As an advantage, use cases are developed only once in the life cycle, through less iterations. Furthermore, the implementation effort for the non-functional code can be fragmented in several points in the phase. However, changing requirements will result in greater impact to the code, since part of the non-functional code will be implemented earlier in the process.

Modifications to the Static Part of RUP

Concerning the static organization of RUP, activities in some RUP workflows were created or modified. The modifications affected the requirements, analysis and design, implementation and test workflows. Although activities from the configuration management workflow are closely related to implementation activities, those activities were not modified, since the configuration management policies must be maintained, even with Pim integrated to RUP.

As the requirements and test workflows are not directly related to Pim activities, they were not significantly changed. In the former

Figure 3: Alternatives for scheduling special iterations



workflow, RUPim includes guidelines on how to write code for user-interface prototypes, stating that if the programmer decides to implement part of functional requirements in those prototypes, he must abstract from non-functional code. This procedure can guarantee the isolation of the functional prototypes from Pim. In the latter workflow, RUPim includes guidelines on designing tests, in order to define different types of tests considering functional and special iterations. The remainder of test activities were maintained as documented on RUP.

On the other hand, the analysis and design and implementation workflows received many modifications for the definition of RUPim, since these workflows are directly affected by Pim's integration to RUP. The main modifications are presented in the following sections.

Analysis and Design

The analysis and design activities suffered two major modifications in order to suggest the use of architectural and design patterns that allow the gradual and separate implementation of different aspects.

First, in the Architectural Design activity, RUPim guides the architect to incorporate the elements from the design patterns from Pim into the architecture of the application. In addition, RUPim suggests how to document the structure and behavior of design elements into architectural mechanisms, which state the relationship between application classes (or subsystems) and design elements from specific platforms.

Second, in the Use-case Design activity, RUPim guides the inclusion of all architectural mechanisms from the Architectural Design activity into the use-case design. This means that the use cases will be designed using elements from Pim's design patterns and elements from specific middleware for implementing the non-functional code.

Implementation

This workflow suffered most of the modifications for defining RUPim. In the Implement Components activity, instead of completely implementing classes and subsystems (as in RUP), programmers should implement only the business and user-interface code, abstracting from non-functional code. The application at this point will be local, sequential and will use in-memory data structures. It can guarantee the correct use of Pim, which guides the initial abstraction of non-functional code in order to minimize the impact of changing requirements.

Besides that, we created three new activities for the implementation of non-functional code for persistence, distribution and concurrence control, following guidelines defined elsewhere [1, 15, 18]. In each of these activities, steps were created in correspondence to Pim tasks for introducing non-functional aspects. In general, these steps address code generation of design classes related to non-functional aspects, implementation of these classes, changes on business classes, code documentation, etc. These activities are performed within special iterations in the RUPim's life cycle.

Practical Experiments

In order to validate RUPim, some practical experiments were performed. In a simple and small-scaled empirical study, use cases from a real web information system were developed twice, in two distinct projects, one guided by a RUP-based methodology, and other guided by a RUPim-based methodology. Both projects were conducted and performed by the same team of three programmers, each one accomplishing by himself one activity at a time.

In the RUP project, three iterations were scheduled (one for elaboration phase and two for construction phase), and the team implemented a complete version of each use case of the application through iterations of the elaboration and construction phases. The RUPim project was decomposed into five iterations (one functional and one special iteration for the elaboration phase, and two functional and one special iteration for construction phase), and the team followed Pim on implementation activities (the alternative of scheduling the special iterations in the end of each phase was adopted). Both projects shared the same design model, otherwise we would change the focus the study, since we would not have an effective way to compare implementation results on both projects.

The two projects were executed sequentially (RUP project first), and quantitative and qualitative data were collected from each implementation and test activity accomplished by a programmer. Based on the results of these experiments, we compared software quality and productivity factors on the two different approaches.

Concerning software quality, we observed that, following RUPim, the effort for changing functional requirements, measured at the point where the complete functionality was released in the two projects, decreased 60%, approximately. Besides that, the effort for performing tests and fixing defects was in general 30% lower, showing that the gradual testing along various intermediate versions of the application helped isolating business problems from aspects problems.

In addition, concerning team productivity, we observed a total productivity gain following RUPim (approximately 11%), due to the lower test effort, even with the fact that programmers in RUPim had to write more code (the test gain was significantly higher than the coding loss). However, using RUPim an application class was edited 50% more times (by successive modifications on the same class, for coding business rules and aspects) and the effort for coding test scripts was 58% higher, approximately (by the testing of intermediate prototypes using in-memory data structures for storing and retrieving data). Future research will focus on specific tools for addressing productivity issues using RUPim, minimizing the main disadvantages of our approach.

It must be stressed that these results do not completely validate the benefits of our approach for industrial-strenght software development. This can only be achieved by performing and analyzing more comprehensive experiments and case studies. In this way we would be able to precisely validate the benefits and identify what must be improved in our new method.

CONCLUSION

We have proposed an extension of the Rational Unified Process (RUP) for supporting the progressive and separate implementation of three different aspects: persistence, distribution and concurrence control code. The resulting software process, RUPim, complements and improves RUP by allowing software teams to achieve the benefits of separation of concerns in industrial development projects that are based on RUP.

When compared to RUP, the new process offers better support to reduce the impact of inevitable requirements changes during development. By following RUPim, functional prototypes are tested and validated continuously, thus most changes will likely occur before implementing the non-functional code, which is implemented latter in the

project. RUPim also helps to tame the complexity of testing distributed and persistent applications, since it allows the gradual and separate test of the application. Our beliefs were validated through simple experiments with the development of use cases from a real application. However, more comprehensive experiments should be performed to better validate our approach.

Several languages and tools for separation of concerns have been proposed [10, 17], but associated processes have received less attention. Instead of using tools or new language constructs, RUPim is based on architectural and design patterns that try to achieve similar results for the three types of non-functional aspects considered here: persistence, distribution and concurrence control. However, as better separation of concerns could be achieved with new language constructs and tools, it would be useful to adapt RUPim to support them as well. We believe that this is possible and useful for large software development projects.

Besides RUP, other modern software processes have been recently proposed [2, 4, 8]. OPEN and OOSP share many aspects with RUP and could be extended with Pim as well, since they do not provide specific implementation methods. We chose RUP because it is widely used in industry nowadays [3]. Extreme Programming (XP), the other alternative, is too focused on implementation activities, not giving much relevance to analysis and design activities and artifacts, for example. However, XP does not give any directions for progressive implementation and it could be extended with Pim as well.

ENDNOTE

1 Partly supported by CNPq, Grant 521994/96-9.

REFERENCES

[1] Vander Alves and Paulo Borba. A Design Pattern for Distributed Applications. In *XIV Brazilian Symposium of Software Engineering — Tutorials*, 4th–6th October 2000.

[2] Scott Ambler. *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.

[3] Scott Ambler. Enhancing the Unified Process. *Software Development Magazine*, September 1999.

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[5] Grady Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison-Wesley, 1999.

[6] Paulo Borba, Saulo Aráujo, Hednilson Bezerra, Marconi Lima, and Sérgio Soares. Progressive implementation of distributed Java applications. In *Engineering Distributed Objects Workshop, ACM International Conference on Software Engineering*, pages 40–47, Los Angeles, USA, 17th–18th May 1999.

[7] James Gosgling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[8] I. Graham, B. Henderson-Sellers, et al. *The OPEN Process Specification*. ACM Press, Addison-Wesley, 1997.

[9] Ivar Jacobson et al. *The Unified Software Development Process*. Object Technology. Addison-Wesley, 1999.

[10] G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of 11th European Conference on Object-Oriented Programming, ECOOP'97*, pages 220–242, June 1997.

[11] Gregor Kiczales et al. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP 2001*, pages 327–353, Budapest, Hungary, 18th–22th June 2001.

[12] Philippe Kruchten. *Rational Unified Process - An Introduction*. Addison-Wesley, 1999.

[13] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections Pattern. In First Latin-american Conference on Pattern Languages of Programming: SugarLoafPLop 2001. October 3rd-5th, 2001. Rio de Janeiro, Brazil. To be published.

[14] David L. Parnas et al. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of ACM*, 15(12):1053–1058, December 1972.

[15] Sérgio Soares and Paulo Borba. Concurrence Control with Java and Relational Databases (in portuguese). In *V Brazilian Symposium of Programming Languages*, 23th–25th May 2001.

[16] Rational Software Corporation. RUP Web Site 2001. http://www.rational.com/products/rup.

[17] Peri Tarr et al. N Degrees of Separation: Multi–dimensional Separation of Concerns. In *1999 International Conference on Software engineering*, pages 107–119. ACM, 1999.

[18] Euricélia Viana. Integrating Java with Relational Databases (in portuguese). Master's thesis, Centro de Informática, UFPE, 2000.

[19] Jim Waldo et al. A Note on Distributed Computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1997.