Progressive Implementation of Aspects

Tiago Massoni   Augusto Sampaio   Paulo Borba*
Centro de Informática
Universidade Federal de Pernambuco

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 concurrency control code. 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.

1 Introduction

Software development has become a more complex activity over the last years. Additional strain results from new common requirements such as distributed and concurrent access. These non-functional issues 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 [14], implementation activities can benefit from this separation.

An implementation methods might help programmers to effectively achieve this separation. Therefore, we have defined the progressive implementation method (Pim) [3], supporting a progressive approach for object-oriented implementation in Java [4], where persistence, distribution and concurrency control aspects are not initially considered in the implementation activities, but are gradually introduced. 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 [9], in such a way that the different aspects can be implemented separately. We believe that the same approach could also be pursued with AspectJ [6], where aspects would play the role played by the patterns in Java.

In this paper we extend the Rational Unified Process (RUP) [7] 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. In Sections 2 and 3, we respectively present the main concepts of RUP and Pim. Section 4 outlines the definition of RUPim, the proposed extension of RUP. Finally, Section 5 presents our conclusions and related work.

2 Rational Unified Process

The Rational Unified Process (RUP) is an industrial software process focused on visual modeling (UML [2]) and three key ideas. First, RUP is use-case driven—use cases, which represent

*Partly supported by CNPq, grant 521994/96-9. Electronic mail: {tlm,phmb}@cin.ufpe.br.

functional requirements in UML, drive the whole development process. Second, RUP suggests the early definition of an stable architecture that supports key use cases, followed by the development of the other use cases of the application, filling the architecture baseline defined early. Third, RUP has an iterative and incremental lifecycle, and each lifecycle iteration develops a set of use cases, representing an increment to the final product [7]. This process can be described in two dimensions:

- The dynamic part of the process, expressed in terms of four serial phases, which are broken down into iterations. In the inception phase the lifecycle 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 static part of the process, 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 lifecycle phases, we choose which activities are performed in each iteration.

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

3 Progressive Implementation Method

The Progressive Implementation Method (Pim) guides the implementation of complex object-oriented applications in Java. Using this method, we do not consider persistence, distribution and concurrency control 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 1 illustrates this progressive approach.

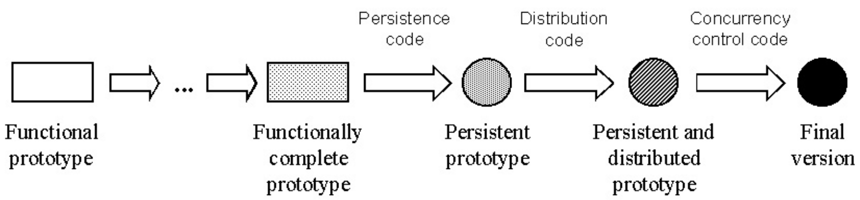

Figure 1: Progressive Implementation Method.

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 [3].

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 concurrency control), this can be achieved with architectural and design patterns [1, 8]. It could also be achieved by using an aspect-oriented programming language [5]. For instance, we could keep persistence related code as an aspect, separately from the business code, and Aspect-J [6] would include persistence to the business code.

4 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). 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.

4.1 Modifications to the Dynamic Part of RUP

Some concepts from RUP's phases and iterations were changed, in order to unify Pim's and RUP's lifecycles properly. 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's collections, for example). The application runs in a single machine, with no concurrent access.

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 concurrency 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.

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. So use cases will be partially implemented in functional iterations, until a functional prototype is finished, and then this prototype will evolve to a persistent and distributed application, with concurrency control, by 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. 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 lifecycle, 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.

4.2 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. As the requirements and test workflows are not directly related to Pim activities, they were not significantly changed. On the other hand, the analysis and design and implementation workflows received several modifications.

4.2.1 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. 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 behaviour of design elements into architectural mechanisms, which state the relationship between application classes (or subsystems) and design elements from specific platforms. 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.

4.2.2 Implementation

This workflow suffered most of the modifications for defining RUPim. The major modifications are the following. 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.

Besides that, we created three new activities for the implementation of non-functional code for persistence, distribution and concurrency control, following guidelines defined elsewhere [1, 10, 12]. In general, these activities address the separate generation of non-functional code and are performed within special iterations in RUPim's lifecycle.

5 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 concurrency control code. The resulting software process, RUPim, complements and improves RUP by allowing us to achieve the benefits of separation of concerns in industrial development projects that are based on RUP. In this work we have discussed the aspects ideas mainly on implementation activities, since Pim is an implementation method. However, in our future work we aim at using techniques and language constructs for considering aspects on design activities, as subject-oriented design [13].

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. Following RUPim, the effort for changing functional requirements and aspects decreased 60%, approximately, and the effort for performing tests and fixing defects was in general 30% lower. However, more comprehensive experiments should be performed to better validate our approach.

Several languages and tools for separation of concerns have been proposed [5, 11], 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 concurrency 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.

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

[3] Paulo Borba, Saulo Araújo, 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.

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

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

[6] 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.

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

[8] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections Pattern. Submitted to the SugarLoafPlop 2001.

[9] 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.

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

[11] 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.

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

[13] J. Vlissides. Subject-oriented design. *C++ Report*, February 1998.

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