PIP: Progressive Implementation Pattern

Sérgio Soares[*] and Paulo Borba[†]
Informatics Center
Federal University of Pernambuco

Intent

Tame complexity and improve development productivity. Reduce the impact caused by requirements changes during development.

Context

When developing a persistent, distributed, and concurrent system, implementation and tests are usually hard. During tests, database, distribution, concurrency, and functional errors might appear at the same time, increasing debugging complexity.

When using EJB [10] as the persistence and distribution technology, the deployment time might be very high. To fix errors — including functional, persistence, and concurrency control errors — we might waste a lot of time by compiling the code and them deploying the system into the application server. Another problem happens when using a database to persist data. We might have to write specific programs to check if the data stored into the database conforms to the expected results. Similarly, if the system can be concurrently accessed, programmers should worry about concurrent executions when implementing functional requirements, increasing programming complexity.

Problem

It is difficult and expensive to validate and test a concurrent, distributed, and persistent system. Furthermore, system validation usually can only be done latter in the development phase. This delay to validate system requirements increases costs to fix detected errors, since developers might dedicate considerable effort to implement non-functional requirements to incorrectly implemented system services.

To implement a persistent, distributed, and concurrent system, *PIP* balances the following forces:

- Early validation of functional requirements. This reduces changes cost and prevents delays in project schedule.
- Simplify tests by testing each aspect (persistence, distribution, and concurrency control) separately. This separation allows testing the functional version of the system without the impact of database, network, or concurrent environments errors. In fact, each non-functional requirement will also be gradually implemented and tested, which avoids that errors of one aspect affects tests of another.

[*]Supported by CAPES. Also affiliated to Catholic University of Pernambuco. Email: `scbs@cin.ufpe.br`

[†]Partially supported by CNPq, grant 521994/96–9. Partially supported by Qualiti Software Processes (www.qualiti.com.br). Email: `phmb@cin.ufpe.br`

- Data storage transparency. This is crucial to initially provide a non-persistent version of the system in order to validate functional requirements without implementing persistence. After that, the system evolves to a persistent version.
- Independence of communication API and middleware. Similar to the persistence aspect, in an early version of the system there is no distribution code, in order to allow early validation of functional requirements. However, the system should evolve to a distributed version, without affecting the requirements already implemented.

Solution

In order to solve the mentioned problem, we should implement functional, persistence, distribution, and concurrency control requirements in a progressive way. In fact, we should first implement the functional requirements, user interface, and non-persistent storage, in order to provide a completely functional prototype, and then implement the others aspects, as illustrated in Figure 1. Although the figure suggests an order for implementing and testing the non-functional requirements, this is not demanded by the process pattern. In fact, *PIP* only requires the different aspects to be implemented and tested in a progressive way.

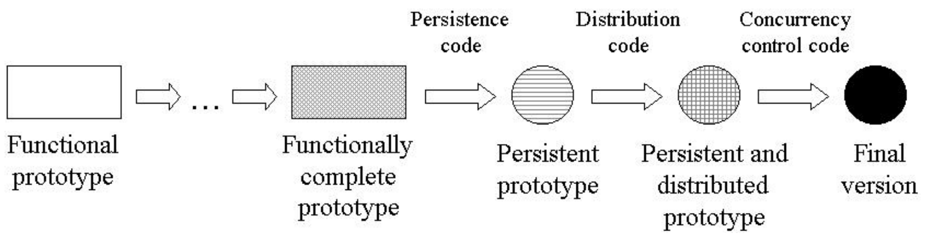

Figure 1: Progressive Implementation Method.

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 the final version of the application, by gradually implementing and testing the persistence, distribution, and concurrency control code.

In order to support this progressive implementation, separation of concerns [11] principles must be applied during design activities. The software architecture must support the modular addressing of functional and non-functional aspects during coding activities. This can be achieved by using specific architectural and design patterns [1, 7, 12].

Alternatively, this separation of concerns can be achieved by using aspect-oriented programming [2]. For instance, we could separate persistence, distribution, and concurrency control aspects from the business code, by using AspectJ [5], an aspect-oriented language, and weave them and the functional prototype into a persistent, distributed, and correct application [14, 13].

Consequences

PIP provides the following benefits:

- *Increased productivity.* Due to the early validation of functional requirements and the simplification of tests, the development productivity is increased. Data collected in a simple case study [9] shows that this increasing is about 10% and there were a 50% reduction on the requirements changes effort. Those numbers can be higher by providing code generation.

- *Tests and debugging are easier.* *PIP* 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, which benefits system correctness.
- *Early functional prototype.* In the simple case study [9] previously mentioned, there is another metric showing that the functional prototype is obtained 30% earlier by using a progressive approach.

This pattern has the following drawbacks:

- *Reduced team motivation.* Programmers might feel that they are generating more code than necessary, for instance, by first generating non-persistent versions of data storage classes and then their persistent versions. To avoid this, the development team should be convinced of the benefits.
- *Limited functional tests.* The progressive approach does not allow to test situations were transactions would be rolled back, with the functional prototype.
- *Additional classes.* When implementing persistence we should create classes to store objects in a persistent medium. However, in order to implement the functional prototype, before implementing persistence, we have to create classes to store the objects in a non-persistence structure. This affects productivity, since programmers should implement two classes to store instances of an object. Code generation tools could solve this drawback by automatically providing part of the implementation of the non-persistent and persistent data storage classes. In fact, even in a non-progressive approach, some non-persistent storage classes should be generated to retrieve data in response to system searches.
- *Additional modifications to classes.* To implement functional requirements, classes are usually modified several times. When using the progressive implementation approach this number increases, since some classes should be modified to implement persistence, then distribution, and finally, concurrency control, decreasing productivity.

Known Uses

Some systems that were developed using this pattern are presented as follows:

- A system to manage clients of a telecommunication company. The system is able to register mobile telephones and manage client information and telephone services configuration.
- A system for registering health system complaints. The system allows citizens to complain about health problems and to retrieve information about the public health system, such as the location or the specialties of a health unit.
- Several small systems developed as undergraduate and graduate projects on object-oriented programming at our institution. Several kinds of systems, such as games, academic control systems, and sales systems, have been developed in these courses.

Besides the mentioned systems that were developed in a progressive way, we can mention some potential uses of the pattern in systems that use the same software architecture and specific design patterns [1, 7, 12] that allow progressive implementation. These systems are the following

- A system for performing online exams. This system has been used to offer different kinds of exams, such as simulations based on previous university entry exams, helping students to evaluate their knowledge before the real exams.
- A complex point of sale system. This system will be used in several supermarkets and is already being used in other kinds of stores.

See Also

- *Use Case Driven Development [3].* This development technique states that system development should be driven by functional requirements. Therefore, developers should create analyzes, design, and implementation models that conform to the functional requirements, and make tests to ensure that the system correctly implement functional requirements. Next section presents how *PIP* interacts with this technique.
- *PDC: Persistent Data Collections [7].* This pattern provides a set of classes and interfaces in order to separate data access code from business and user-interface code, promoting modularity. The pattern defines a structure to archive storage transparency. This structure allows implementing persistence after the functional requirements implementation.
- *DAP: Distributed Adapters Pattern [1].* This pattern provides a structure for implementing remote communication between two components, decoupling them from specific communication Application Programming Interface (API). This pattern's structure also allows implementation of distribution after the functional requirements implementation.
- *PaDA: A Pattern for Distribution Aspects [13].* This pattern is similar to DAP in the sense that provides a structure for implementing distribution code. However, PaDA achieves better separation of concerns, through the use of aspect-oriented programming (AOP) [2].
- *Concurrency Manager [12].* This pattern provides an alternative to method synchronization with the aim of increasing system performance. *Concurrency Manager* uses knowledge about the semantics of the methods in order to block only conflicting execution flows, allowing the non–conflicting ones to execute concurrently. It can be used to improve concurrency control.

There are variations of *PDC* and *DAP* that use EJB to implement persistence and distribution [6].

Interactions with other patterns

Based in RUPim [8, 9], a RUP extension that defines how to extend the Rational process with the Progressive implementation method (Pim), this section describes how *PIP* interacts with *Use Case Driven Development* [3], a well know and used development technique, which is used by the Rational Unified Process (RUP) [4] and other processes. In fact, we suggest this kind of section to be added to the process patterns template, in order to explicitly describe how the pattern interacts with other process patterns. As design patterns have a well-defined structure, it is easier to understand how they interact with each other. We think that a major challenge for the widespread use of process patterns is to dearly define how they depend on and interact with each other. Most of the related patterns are actually design patterns that are necessary to supporting the use of the Progressive Implementation Process Pattern.

A use case defines what interactions occur between a system and its users, capturing system requirements. The use cases of a system constitute a use case model. In *Use Case Driven*

Development (*UCDD*), developers create design and implementation models that realize the use cases. Moreover, other models should comply with the use case model, and tests should ensure that the use cases are correctly implemented.

In order to combine *UCDD* with *PIP*, providing a use case driven progressive development, we should define how and when non-functional requirements are to be considered and implemented. In *UCDD* a system is designed, implemented, and tested based on its use cases. When considering a progressive implementation, design models should favor the progressive implementation, as mentioned in the forces of the Section Problem.

To implement a use case, programmers should implement parts of the system that are necessary to realize the use case. However, when planning development combining *UCDD* with *PIP*, non-functional requirements implementation should be schedule after implementing the functional part of the use cases and the user interface code. Therefore, use cases will be partially implemented in functional iterations, until a functional prototype is finished. At this moment, this prototype should be validated and, if necessary, changes should be made. After validating the implemented functional code, the prototype will evolve to a persistent and distributed application, with concurrency control.

Another alternative to combine *PIP* with *UCDD* is to plan interchanged functional and non-functional implementation during use case implementation. Contrasting with the first alternative, use cases are completely implemented, in their corresponding functional and non-functional requirements implementation activities. As an advantage, use cases are developed only once in the lifecycle. Furthermore, the implementation effort for the non-functional code can be fragmented in several points. 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, also increasing tests complexity.

References

[1] Vander Alves and Paulo Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[2] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect–Oriented Programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[3] Ivar Jacobson. Object-oriented development in an industrial environment. In *Proceedings of the OOPSLA'87 conference on Object-oriented programming systems, languages and applications*, pages 183–191. ACM Press, December 1987.

[4] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[5] Cristina Lopes and Gregor Kiczales. Recent developments in AspectJ. *Workshop on Aspect–Oriented Programming at ECOOP'98*, July 1998.

[6] Klissiomara Lopes and Paulo Borba. Design Patterns to Structure Enterprise JavaBeans Distributed Applications (in portuguese). In *Second Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, August 2002.

[7] Tiago Massoni, Vander Alves, Sérgio Soares, and Paulo Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[8] Tiago Massoni, Augusto Sampaio, and Paulo Borba. Progressive Implementation of Aspects. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems — OOPSLA'01*, Tampa Bay, USA, 14th-18th October 2001.

[9] Tiago Massoni, Augusto Sampaio, and Paulo Borba. A RUP-based Software Process Supporting Progressive Implementation. In Idea Group Publishing, editor, *2002 Information Resources Management Association International Conference (IRMA 2002)*, pages 480–483, Seattle, USA, 19th-22nd May 2002.

[10] Richard Monson-Haefel. *Enterprise JavaBeans*. Oreilly, second edition, 2000.

[11] David L. Parnas et al. On the criteria to be used in decomposing systems modules. *Communications of the ACM*, 15(12):1053–158, December 1972.

[12] Sérgio Soares and Paulo Borba. Concurrency Manager. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[13] Sérgio Soares and Paulo Borba. PaDA: A Pattern for Distribution Aspects. In *Second Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, August 2002.

[14] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA'02, Object Oriented Programming Systems Languages and Applications*. ACM Press, November 2002. To appear.