Aspect-Oriented Implementation Method: Progressive or Non-progressive Approach?

Sérgio Soares*
Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil
scbs@cin.ufpe.br

Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil
phmb@cin.ufpe.br

ABSTRACT

Object-oriented programming languages provide effective means to achieve better reuse and extensibility levels, which increases development productivity. However, the object-oriented paradigm has several limitations, sometimes leading to tangled code and spread code. For example, business code tangled with presentation code or data access code, and distribution, concurrency control, and exception handling code spread over several classes. This decreases readability, and therefore, system maintainability. Some extensions of the object-oriented paradigm try to correct those limitations allowing reuse and maintenance in practical situations where the original paradigm does not offer an adequate support. However, in order to guarantee that those benefits will be achieved by those techniques it is necessary to use them together with an implementation method. Our objective is to adapt and to analyze an object-oriented implementation method to use aspect-oriented programming in order to implement several concerns to a family of object-oriented system. In particular, we are interested in implementing persistence, distribution, and concurrency control aspects. At the moment we are particularly interested to present some results and get feed back about a performed experiment to identify if and when a progressive approach is better than a non-progressive one. In a progressive approach, persistence, distribution, and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the application's functional requirements. This approach helps in dealing with the inherent complexity of the modern applications, through the support to gradual implementations and tests of the intermediate versions of the application.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—Aspect-Oriented Programming; D.3.2 [**Programming Languages**]: Language Classifications—AspectJ

*Also affiliated to Catholic University of Pernambuco, Informatics and Statistics Department, Recife, Pernambuco, Brazil.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

General Terms

Languages, Standardization, Experimentation

Keywords

Aspect-oriented programming, separation of concerns, AspectJ, Java, implementation method, progressive implementation

1. INTRODUCTION

Usually, researchers and software engineers do not give much attention to implementation methods [1, 6], because implementation mistakes have less impact in project schedule and development costs than mistakes regarding requirements and design.

However, the effort given to requirements and design can be wasted if there is not a commitment with the implementation activity. This is necessary in order to increase productivity, reliability, reuse, and extensibility levels. For example, the maintenance activity usually has the highest cost [3, 5], which is inversely proportional to reuse and extensibility. This motivates the continuous search to increase those levels.

Object-oriented programming languages provide effective means that help to increase productivity, reliability, reuse, and extensibility levels, but has several limitations, sometimes leading to tangled code and spread code, decreasing readability, and therefore, system maintainability. Examples are, business code tangled with presentation code or data access code, and distribution, concurrency control, and exception handling code spread over several classes. To solve these limitations, techniques, like aspect-oriented programming, aim to increase software modularity in practical situations where object-oriented programming does not offer an adequate support.

2. ASPECT-ORIENTED PROGRAMMING

We believe that aspect-oriented programming (AOP) [4], is very promising [9, 11]. AOP tries to solve the inefficiency in capturing some of the important design decisions that a system must implement. This difficulty leads the implementation of these design decisions spread through the functional code, resulting in tangled code with different concerns. This tangling and scattering code hinders development and maintenance of these systems. AOP increases modularity by separating code that implements specific functions and

affects different parts of the system. These are called crosscutting concerns.

By separating concerns AOP allows implementing a system separating functional and non-functional requirements. For example, a set of components written in an object-oriented programming language, such as Java, might implement functional requirements. On the other hand, a set of aspects (crosscutting concerns) related to the properties that affect system behavior might implement non-functional requirements. Using this approach, non-functional requirements can be easily manipulated without impacting the business code (functional requirements), since they are not tangled and spread over the system. In this way, AOP allows the development of programs using such aspects, including isolation, composition and reuse of part of the aspects code.

3. IMPLEMENTATION APPROACHES

No matter how good the programming language, an implementation method is important to define activities to be executed and the relations between them, including their execution order. Our main goal is to define an implementation method using aspect-oriented programming, helping to develop better software with better productivity levels. Our implementation method will guide the implementation of persistence, distribution, and concurrency control concerns that conforms to specific software architecture. Despite being specific, the software architecture can be used to implement several kinds of systems.

These aspects can be implemented in different ways and in a different order. They might be implemented at the same time as the functional requirements are being implemented. Another idea is to follow a progressive approach, where persistence, distribution, and concurrency control are not initially considered in the implementation activities, but are gradually introduced, preserving the system's functional requirements.

This progressive approach helps in decreasing the impact in requirements changes during the system development, since a great part of the changes might occur before the final version of the system is finished. This is possible because a completely functional prototype is implemented without persistence, distribution, and concurrency control, allowing requirements validation without interference of these non-functional requirements and without the effort to implement those. At this time the system uses non-persistent data structures, such as arrays, vectors, and lists, and is executed in a single-used environment. Moreover, the progressive approach helps in dealing with the inherent complexity of modern systems, through the support to gradual implementation and tests of the intermediate versions of the system.

3.1 Approaches analysis

We performed an experiment with graduate students using AspectJ [7] and the implementation approaches to identify if and when the progressive approach is better than the non-progressive one. The experiment was carefully designed using recommendations of experts in the empirical area [10, 8, 2].

We divided the students in pairs and randomly assigned to a project. There were two kinds of project; both had the same resulting system, however one had to follow one a progressive approach, and the other a non-progressive approach. In the experiment execution they implemented a simple information system with operations to register, change, and retrieve information. We simulate development problems like requirement changes and modeling problems and we also simulate code generation to support the development. An interesting result of this experiment was new interferences between the aspects that were not identified in the previous experiment.

In this experiment we collected data in order to evaluate the benefits and liabilities to implement a system using a progressive approach. Examples of the data collected are implementation time, debugging time, time to correct errors and requirements changes, number of lines affected by changes, and a form to get the anonymous feedback about the experiments from the students. The experiment execution is already finished, however we had some problems during data collection, which did not allow we to use some of the values. The others values showed that the progressive approach maybe more effective in those experiments. However, we did not have enough groups to provide statistical analysis, we only used data of two groups. Therefore, we are planning to run the experiment again taking care of how the data will be collected allowing us to validate this previous experiment.

4. REFERENCES

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

[2] Victor Basili, Richard Selby, and David Hutchens. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, SE-12(7):733–743, July 1986.

[3] D. Coleman, D. Ahs, B. Lowther, and P. Oman. Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*, 24(8):44–49, August 1994.

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

[5] R. Graddy. Succesfully Applying Software Metrics. *IEEE Computer*, 27(9):18–25, September 1994.

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

[7] Gregor Kiczales, et. all. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[8] Gail Murphy, et. all. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, July/August 1999.

[9] Gail Murphy, et. all. Does aspect–oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.

[10] Shari Pfleeger. Design and Analysis in Software Engineering. *Software Engineering Notes*, 19(4):16–20, October 1994.

[11] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *OOPSLA'02*, pages 174–190. ACM Press, November 2002. SIGPLAN Notices 37(11).