Towards Progressive and Non-progressive Implementation Approaches Evaluation

Sérgio Soares*
Universidade Católica de Pernambuco
Departamento de Estatística e Informática
Recife, Pernambuco, Brasil
sergio@dei.unicap.br

Paulo Borba
Universidade Federal de Pernambuco
Centro de Informática
Recife, Pernambuco, Brasil
phmb@cin.ufpe.br

ABSTRACT

Unfortunately, there is a lack of experiments in software engineering. For example, it is common to propose new processes, paradigms, methods, development architectures, frameworks, and others, using only intuitive or textual arguments, instead of trying to demonstrate effectiveness of the proposal by using experiments techniques, when real case studies might be to expensive. We are particularly interested to present some results and get feed back about a performed study to identify if and when a progressive implementation 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.

Keywords

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

1. INTRODUCTION

Usually, researchers and software engineers do not give much attention to implementation methods, 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, 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, software 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.

*Work performed while the first author was at CIn/UFPE.

2. ASPECT-ORIENTED PROGRAMMING

We believe that aspect-oriented programming (AOP) is very promising [5]. AOP tries to solve the inefficiency in capturing some of the important design decisions that a software 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 software development and maintenance. AOP increases modularity by separating code, called crosscutting concerns, that implements specific functions and affects different parts of the software.

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. We defined an implementation method [4] using aspect-oriented programming, helping to develop better software with better productivity levels. The implementation method guides the implementation of persistence, distribution, and concurrency control concerns that conforms to a specific software architecture. Despite being specific, the software architecture can be used to implement several kinds of software.

These crosscutting concerns 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 to deal with the inherent complexity of modern systems, through the support to gradual implementation and tests of the intermediate system versions.

3.1 Approaches analysis

We performed an experiment with graduate students using AspectJ and the implementation approaches to identify if and when the progressive approach is better than the non-progressive one, using recommendations of experts in the empirical area [1, 3]. We divided the subjects in two groups 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 faults like requirement changes and modeling problems. We also simulate code generation to support the development. Examples of data collected in the experiment are:

- Time to implement selected use-cases;
- Time to perform changes in selected use-cases, after the client or the software architect request the change;
- Time to run test-cases of selected use cases;
- Time to yield a first executable prototype of selected use cases, before validation;
- Time to yield an executable prototype validated by the client. This implies that requirements were validated and some of them might have changed. A mentor played the client role.

3.2 Results

We used a t-test [2, 6] to make data analysis and therefore, hypothesis testing. We also analyzed the collected data by determining the confidence interval [2] for the mean difference of progressive and non-progressive effort. If the interval contains zero, which means that zero is a possible value for the difference, the samples are not significantly different. This might be a more effective way to analyze data than just saying if the samples mean are different or not. A narrow confidence interval indicates a high degree of precision. On the other hand, a wide confidence interval indicates that the precision was not high.

We first used hypothesis testing to identify if the samples are significantly different. When the null hypothesis is false, the sample means are significantly different and it is not necessary any additional analysis. On the other hand, when the null hypothesis is true, instead just saying that the sample means are not significantly different, we also determine the confidence interval to analyze if the degree of precision of this analysis.

According to the presented results, at all iterations, implementation and tests time using a progressive approach are not significantly different from the non-progressive approach. On the other hand, the statistical test rejected the null hypotheses for the time to change requirements, and yield pre and post-validation prototypes. The confidence interval for implementation and test time indicated that the null hypotheses test were obtained with a low precision. In fact, we expected benefits from implementing and testing the software with the progressive approach. When progressively implementing and testing functional requirements, the programmer does not have to worry with persistence and distribution issues and vice-versa, decreasing implementation and test complexity. These wide confidence intervals suggest that others studies should be performed to better evaluate the progressive approach impact in implementation and tests.

Another important analysis result was about the times to yield pre and post-validation prototypes. The t-test showed that the times to yield pre and post-validation prototypes of all use-case scenarios were significantly different. In fact, the progressive approach had unbeatable results, which was expected since early validation of functional requirements is one of the pillars of the progressive approach. This is reached by first abstracting the implementation of some non-functional requirements. This early validation anticipates requirements change, also helping to understand the problem before implementing some non-functional requirements. Moreover, the effort to create such prototype is lower, decreasing the budget impact, for example, if the requirements were not well understood by the requirements engineering or the clients had just change his mind.

After finishing the study, we applied a questionnaire with general questions in order to get some feedback from the subjects about the study, the technology used, and the implementation approaches. The subjects gave important feedback, such as, 92% said that AOP and AspectJ helped the development and 69% felt that the progressive approach increases productivity. On the other hand, 85% reported some difficulty to learn a new paradigm (AO).

More about the experiment and the collected data can be found elsewhere [4].

4. REFERENCES

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

[2] Raj Jain. *The art of computer systems performance analysis: techiniques for experimental design, measurement, simulation and modeling.* Wiley, 1991.

[3] Shari Pfleeger. Design and Analysis in Software Engineering, Part 1: The Language of Case Studies and Formal Experiments. *Software Engineering Notes*, 19(4):16–20, October 1994.

[4] Sérgio Soares. *An Aspect-Oriented Implementation Method.* PhD thesis, Informatics Center, Federal University of Pernambuco — CIn-UFPE — Brazil, October 2004. To appear.

[5] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, OOPSLA'02*, pages 174–190. ACM Press, November 2002. Also appeared in ACM SIGPLAN Notices 37(11).

[6] C. Wohlin, P. Runeson, M. Höst, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.