International Journal of Software Engineering
and Knowledge Engineering
Vol. 25, Nos. 9 & 10 (2015) 1753–1758
© World Scientific Publishing Company
DOI: 10.1142/S0218194015710126

World Scientific
www.worldscientific.com

Research Notes on the Architectural Evolution of a Software Product Line

Marcelo Schmitt Laser*, Elder Macedo Rodrigues†,
Anderson Domingues‡, Flavio Oliveira§ and Avelino F. Zorzo¶

School of Computer Science
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS 91330-630, Brazil
*marcelo.laser@acad.pucrs.br
†elder.rodrigues@pucrs.br
‡anderson.domingues@acad.pucrs.br
§flavio.oliveira@pucrs.br
¶avelino.zorzo@pucrs.br

This work presents an experience report on the architectural decisions taken in the evolution of a Software Product Line (SPL) of Model-based Testing tools (PLeTs). This SPL was partially designed and developed with the intention of minimizing effort and time-to-market during the development of a family of performance testing tools. With the evolution of our research and the addition of new features to the SPL, we identified limitations in the initial architectural design of PLeTs' components, which led us to redesign its Software Product Line Architecture (SPLA). In this paper, we discuss the main issues that led to changes in our SPLA, as well as present the design decisions that facilitate its evolution in the context of an industrial environment. We will also report our experiences on architecture modifications in the evolution of our SPL with the intention of allowing easier maintenance in a volatile development environment.

Keywords: Software product lines; factory method; architecture evolution.

1. Introduction

Over a few decades, more and more software development companies have been using some software engineering strategies, such as reuse-based software engineering, to develop software with less cost, faster delivery and increased quality. Reuse-based software engineering is a strategy in which the development process is focused on the reuse of assets and on a core architecture, reducing the development effort and improving the software quality. In recent years, many techniques have been proposed to support software reuse, such as Component-based development and Software Product Lines (SPL) [8]. Component-based development is centered on developing a software system by integrating components, where each component can be defined as an independent software unit that can be used with other components to create a system module or even a whole software system. In another way, Software Product Lines are focused on developing a family of applications based on a common architecture and a shared set of software assets, where each application is generated, in accordance with the requirements imposed by different customers, from these assets and shares a common architecture [8]. One of the main SPL development sub-processes is to use the domain requirements and the product line variability model to define the Software Product Line Architecture (SPLA). The SPLA is a common, high level and generic structure that will be used for all the products derived from the SPL. In order to take advantage of this approach, we have adopted the SPL concept to support the development of our applications [1]. We have found that, although this enabled the reuse of artifacts, thus reducing the time and cost of development, it incurred in a cost related to the evolution of each artifact, as well as that of the SPLA used to manage this evolution.

In this work we report and discuss our experience in implementing and evolving the architecture of a component-based SPL to derive Model-based Testing (MBT) tools.[a] In particular, we describe how we applied software design patterns [4] to map and to instantiate components, these having their variability managed by another component. Finally, we describe two methods of implementing the variable components (features).

This paper is organized as follows. Section 2 describes the context where PLeTs SPL was designed and developed, as well as briefly presenting its Product Line Architecture (PLA). In Sec. 3 we discuss the main PLA limitations identified along the SPL evolution. In Sec. 3.1 we present and discuss our approach to mitigate these limitations, as well as describe our PLA in accordance with that approach and in Sec. 4 we discuss the related work. Section 5 presents the lessons learned along the PLA evolution and also the conclusion and future work.

2. Context

Our research group on Software Testing has been working to design and develop several testing tools for the past years. Our research focus is to investigate innovative ways to mitigate the effort of repeatedly creating custom solutions to apply performance, functional and structural testing.[b] After developing several testing tools, either from scratch or using a limited opportunistic reuse, but which had several features in common, we started a collaborative study with the Technology Development Lab (TDL) of our partner company to investigate the use of SPL concepts to generate these testing tools. This TDL often created custom components to enable the testing of non-trivial applications. To eliminate the effort of repeatedly creating

[a] A more complete description can be found in [6].

[b] Study financed by Dell Computers of Brazil Ltd. with resources of Law 8.248/91.

custom infrastructure, the TDL started considering the adoption of SPL concepts. As a result, we consolidated our SPL called PLeTs [1, 7, 6].

PLeTs was initially designed to support the derivation of a particular testing tool from a set of shared software components, which are then glued together with minimal changes. We defined the use of a replacement mechanism to develop each concrete feature [9] of the PLeTs feature model [1]. In this way, an MBT tool derived from PLeTs is assembled by selecting a set of components and a common software base. We chose this approach to generate PLeTs products because it presents some advantages, such as high-level of modularity and a simple 1:1 feature to code mapping. Since we are using a component replacement mechanism, each provided interface represents a variation point and each variable component implementation represents a variant.

3. PLeTs Architecture

In the early versions of PLeTs [1, 3, 7], we have attempted to solve the problem of SPLA volatility with two different approaches: *Component Interfaces* to map the access between components and, *compile-time definitions* to isolate statements that instantiate variability [2].

Neither the use of Component Interfaces, nor the use of compile-time definitions fully addressed the difficulties we found in evolving our SPLA. Due to our research center environment being volatile (team members regularly moving from one project to another), we found it necessary to establish certain guidelines and mechanisms for the maintenance and extension of the SPLA.

3.1. *PLeTs architecture evolution*

In order to tackle the issues raised by our previous development paradigms, we have adopted a mixed approach that is largely based on the use of the Factory Method design pattern [4] to externalize variability points from the implementation of concrete features [9]. We kept the component-based approach. The difference is that our SPL now has a well-defined core that centralizes the variability management, as well as serving as a starting point to the execution of any derived product. The core of our SPL, which is depicted in Fig. 1 (a more detailed version can be found in [6]), is composed by four components:

(1) **Control Unit:** The Control Unit component is responsible for orchestrating the execution of the system, providing access to the functions of those components that implement features and organizing the data structures necessary for the proper execution of the system. It is designed and implemented without any dependencies on components external to the SPL core, which protects it from modifications in them.

(2) **Factory Interfaces:** In order to access the components that are external to the core, that is, all features, the Control Unit makes use of the Factory Interfaces component, which is an abstract representation of the variability points of the SPLA.

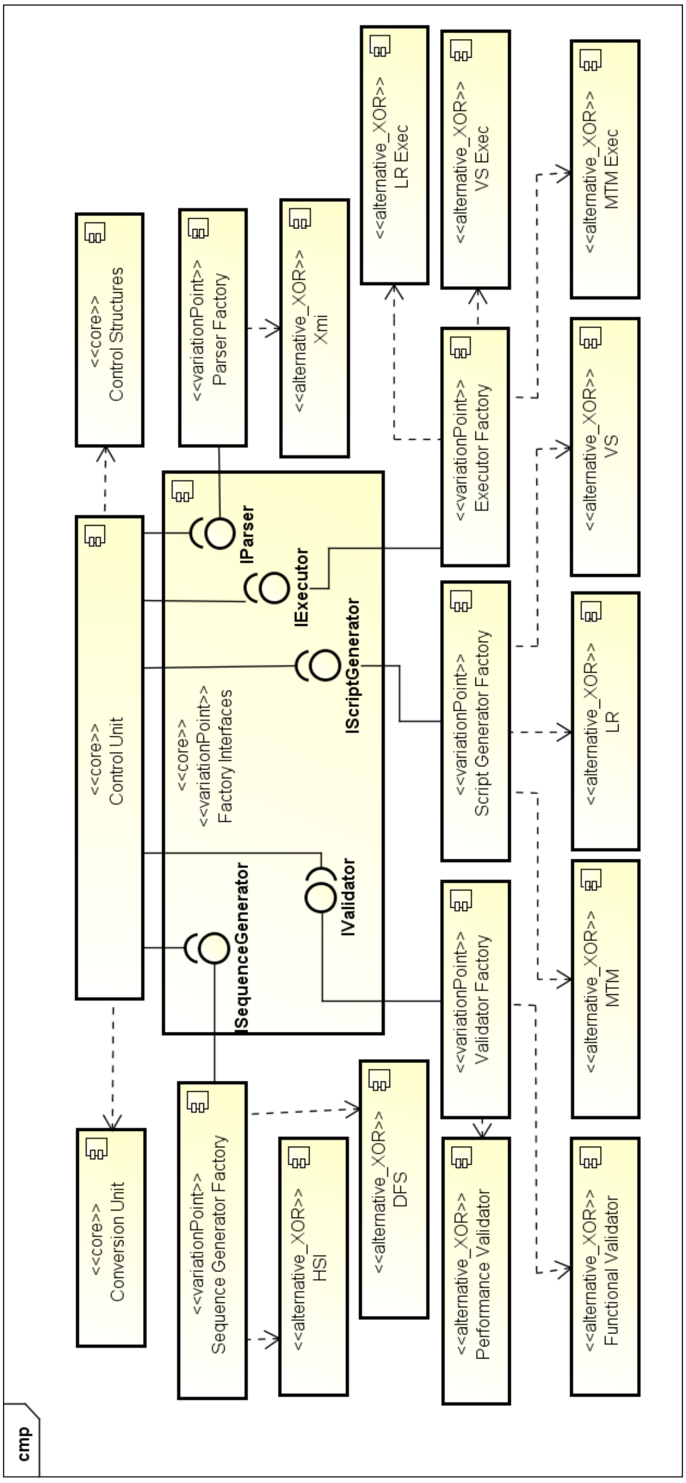
cmp
<<core>> Control Structures
<<variationPoint>> Parser Factory
<<alternative_XOR>> Xmi
<<alternative_XOR>> LR Exec
<<alternative_XOR>> VS Exec
<<alternative_XOR>> MTM Exec
<<core>> Control Unit
<<core>> <<variationPoint>> Factory Interfaces
IParser
IExecutor
IScriptGenerator
IValidator
ISequenceGenerator
<<variationPoint>> Executor Factory
<<alternative_XOR>> VS
<<variationPoint>> Script Generator Factory
<<alternative_XOR>> LR
<<alternative_XOR>> MTM
<<variationPoint>> Validator Factory
<<core>> Conversion Unit
<<variationPoint>> Sequence Generator Factory
<<alternative_XOR>> HSI
<<alternative_XOR>> DFS
<<alternative_XOR>> Performance Validator
<<alternative_XOR>> Functional Validator

Fig. 1. PLeTs improved - UML component diagram.

It contains interface definitions for each variability point, each serving as a connection point for a variable component. Each variability point in the SPL Architecture is represented here by one interface.

(3) **Control Structures:** To access the data structures held by the components external to the core, the Control Unit makes use of the Control Structures component, containing any representations that are common to two or more data structures of the system.

(4) **Conversion Unit:** The Conversion Unit is responsible for parsing structures that are equivalent, i.e. any parsing process that does not change the content of a structure, such as the refactoring of a structure to execute different functions or the updating of a structure to a newer version. For every abstract structure defined in the Control Structures component, the Conversion Unit has a factory capable of reading its type, as well as the return type desired, and forwarding it to the appropriate concrete structure converter. If a new structure component is developed for the system, specific converters will have to be implemented for that structure in order to convert both to and from it. All of these converters are contained within the Conversion Unit component itself, and are therefore accessible by the SPL Core.

(5) **Variable Components:** To allow each feature component to be developed autonomously, an intermediate entry point is represented in the SPLA in the form of Factory components. More about the Factory Method design pattern can be found in [4].

4. Related Work

The authors of [10] speak extensively on techniques for Variability Management and present the case study of the Mercure PL, in which the Abstract Factory design pattern is used as a decision model, with each of its concrete factories being related to one product. Our approach has similarities with the one presented in this particular work, but diverges from it in that our use of the Factory Method design pattern is extended to deal with each of the variability points of the SPL. This is because, while in Mercure PL the authors were deriving whole products, we are applying the Factories to each variation point, requiring a lower level of abstraction.

In [5], this topic is also discussed. A two-dimensional model is proposed for the representation of the issues in variation management, with "files", "components" and "products" in one axis and "sequential time", "parallel time" and "domain space" in the other. The author argues that the nine smaller issues defined by this model can be tackled using a divide-and-conquer strategy.

5. Conclusion and Lessons Learned

In this paper we report our a summary of our experience on the design, development and evolution of a Software Product Line of Model-based Testing tools - PLeTs [6].

We have focused on techniques to simplify the process of managing the evolution of components by use of a software design pattern.

The lessons learned from the development and evolution of PLeTs and the subsequent evolution of its SPLA are: (1) to mitigate the problem of chain modifications and maintenance in components, we have proposed an SPL core that contains all the basic operations supported by the SPL; (2) in using the Factory Method, we were able to isolate the code referent to the majority of variability decisions into small sections that are easy to maintain; (3) connecting a new component to the SPL through one of the pre-existing factory interfaces is a simple process, requiring only the packaging of input and output in accordance to the Control Structures component of the core; (4) if changes in requirements result in new variability points in the SPLA, thus the SPL core requires modifications; (5) as a future work, we plan to conduct a study on the evaluation of the effectiveness of this proposal in an industrial environment.

References

1. L. T. Costa, R. Czekster, F. M. Oliveira, E. M. Rodrigues, M. B. Silveira and A. F. Zorzo, Generating performance test scripts and scenarios based on abstract intermediate models, in *24th SEKE*, 2012, pp. 112–117.
2. K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications* (ACM Press/Addison-Wesley, 2000).
3. E. de Macedo Rodrigues, L. Passos, F. Teixeira, A. F. Zorzo and R. S. Saad, On the requirements and design decisions of an in-house component-based SPL automated environment, in *26th SEKE*, 2014, pp. 483–488.
4. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley–Longman, 1995).
5. C. W. Krueger, Variation management for software production lines, in *Software Product Lines*, 2002, pp. 37–48.
6. M. S. Laser, E. M. Rodrigues, A. R. P. Domingues, F. M. Oliveira and A. F. Zorzo, Architectural evolution of a software product line: An experience report, in *27th SEKE*, 2015, pp. 1–6.
7. M. B. Silveira, E. M. Rodrigues, A. F. Zorzo, H. Vieira and F. Oliveira, Model-based automatic generation of performance test scripts, in *23rd SEKE*, 2011, pp. 1–6.
8. I. Sommerville, *Software Engineering* (Pearson/Addison–Wesley, 2011).
9. T. Thum, C. Kastner, S. Erdweg and N. Siegmund, Abstract features in feature modeling, in *15th Int. Soft. Product Line Conference*, 2011, pp. 191–200.
10. T. Ziadi, J.-M. Jézéquel, F. Fondement *et al.*, Product line derivation with UML, in Software Variability Management Workshop, Univ. of Groningen Department of Mathematics and Computing Science, 2003.