Case Study: Refactoring of Software Product Line Architecture - Feature Smells Analysis

Zdravko Roško
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
zrosko@gmail.com

Abstract. *Software Product Line (SPL) architecture refactoring is typically performed to keep pace with changing environment, such as client platforms, operating system, language compilers, development tools, external third party components and database managements systems. Product Line Architecture (PLA) is a shared architecture for a set of closely related applications. In this paper we report the experience conducting a case study on PLA refactoring analysis to be used as an input to the next stage within the process of its refactoring. Quantitative data are collected from a product line for business applications in a financial institution. The overall goal of the case study was to understand the current characteristics of the PLA with the intention of improving it and making its necessary adaptive and preventive maintenance changes. We propose a refactoring analysis steps for product-preserving type of product line refactoring to ensure improved PLA quality attributes.*

Keywords. Software product lines, refactoring, business applications, features smells, feature.

1 Introduction

One of the most successful approaches to planned and proactive reuse of software assets is Software Product Lines (SPL) approach. SPL is defined as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1]. A software product line (or software product family) relies on a common product line architecture (also called reference architecture) to achieve a substantial increment in product cost, quality and a time to market. Product Line Architecture (PLA) is a base for all components that are used by individual product in the product line. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change [2]. Therefore, the reference architecture development is a key activity for organizations following a SPL approach since this core asset allows to keep pace with changing environment and with market's present and future needs. Specifics of product line refactoring differ from refactoring in general, which is usual practice in software development. To give additional insights into product line refactoring specifics, we summarize experience from a refactoring project we performed using FORM (Feature-Oriented Reuse Method [3]) process method.

2 Related work

Refactoring in general, not specific to product lines, are typically traced to the dissertation of Opdyke [4]. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure [5]. A good source of refactoring in general in object-oriented programming is the book by Flower [5]. However, in the context of product lines, this definition is not sufficient, because it does not take into account a whole family of applications and their relationship with the common reference architecture. Alves et al. were among the first to propose to extend traditional notion of refactoring to software product lines [6]. Thum et al. proposed an automated analysis to identify refactoring on feature models [7].
In this paper, we use extended traditional notion of refactoring, in which SPL refactoring is a change made to the structure of a SPL in order to improve (maintain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behavior of its original products.

3 Case and study selection

The subject of this study was an implementation of PLA for business applications in a financial institution. There are 7 business applications using the same PLA as a base to provide the required user functionality to the business users of the institution. Examples of such applications include loan processing application that allow users to process a loan request, credit card scoring application that allows users to calculate the application score for a new credit card application, etc. PLA consists of three large subsystems as shown in Figure 1: Client for Java
applications, Shared components used on client and on server side of the applications, and Server which consists of Business Logic and Server Persistence layers. The starting version of the reference architecture is based on Java 1.5 and the target refactored Java platform is Java 1.7. Also, the starting version support Java client applications only, but target version should support web based Ajax enabled client applications. Since Java 1.7 supports many of the valuable language features which may help to improve the quality attributes of reference architecture, our domain analysis include the capabilities of the proven programming approaches and techniques such as: Aspect Oriented Programing (AOP), Annotations, Inversion of Control (IoC), and some others.

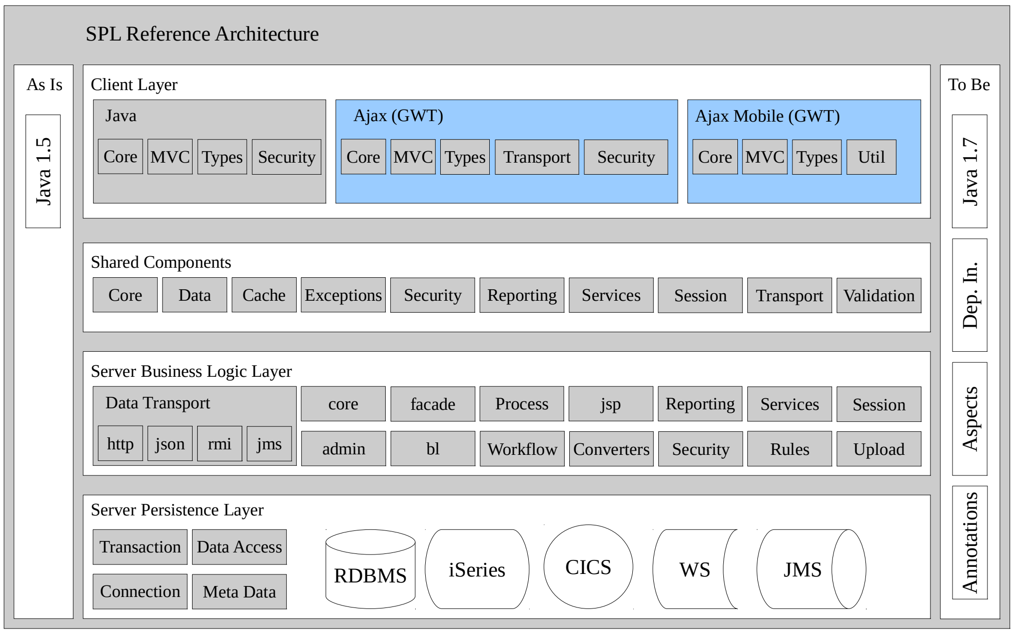


Figure 1. PLA for Business Applications

Refactoring of a product line is an important activity which needs to be planned since the changes performed reflects not only a single application, but a set of artifacts that may be used to generate a whole family of applications. The study we present here reflects the refactoring performed as a reaction to a feature smell, a perceived problem in the source code, which belong to an *adaptive* and *preventive* category of software change [8]. *Adaptive* changes are made in reaction to a changing environment such as new language compilers (e.g. Java 1.7), new operating system, third party external components, database management system, integrated development environment and tools, etc. *Preventive* changes are made to improve future maintainability and reliability of the product line components. Unlike adaptive reason for change, preventive changes proactively seek to improve quality attributes of the future product line applications and the reference architecture components.
Overall, the central focus of this study is the improvement of PLA maintenance quality attributes, especially changeability and stability. Changeability measures the impact made to a component of product line to the rest of the reference architecture components and related product line applications. Increasing number of external third party components dependency which are referenced by product line reference architecture components or applications periphery components, impacts the stability of product line. Changes to the external third party components is a threat to the stability of a product line. These changes should be addressed by delegating the responsibility of the changes to the reference architecture rather than leaving them to be handled by product line applications separately.
Product line includes core assets (reference architecture components and application components) and individual applications (products) composed from those core assets. Synchronizing the release of features and components in core assets with the product releases is a key to managing product line. The release of a product at a given point in time requires that core

assets used by product meet or exceed the quality and functionality for that product release. Figure 2 illustrates the synchronized product line release generations (Baseline 1 through Baseline Refactored) occurring on annually intervals [9]. In between there were monthly synchronization points, based on user requirements or maintenance changes. Time is shown across the top horizontal axis. At the bottom of the figure are the applications (products). Note that during three years the number of products increases. The refactored baseline, the last release of the baseline is the final ("to-be") release of the product line, which we target by this refactoring activity.

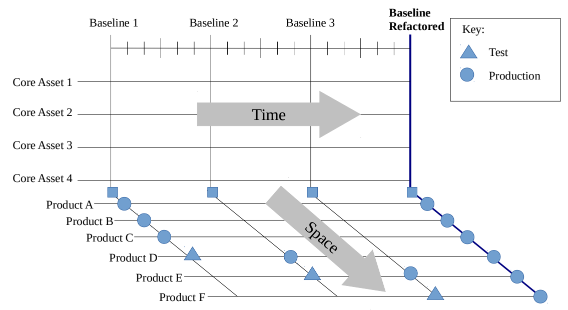


Figure 2. SPL releases

4 Analysis procedure

We have selected to use the FORM, an architecture design process to serve as basis for obtaining Baseline Refactored release. FORM is a systematic method that looks for and captures commonalties and differences of applications in a domain in terms of "features" and using the analysis results to develop domain architectures and components. The model that captures the commonalties and differences is called the "feature model" and it is used to support both engineering of reusable domain artifacts and development of applications using the domain artifacts [3]. Applications users and software developers are both familiar with use of the term "features" when communicate a product characteristics in terms of "features the product has or need to have". FORM is based on a commonality analysis expressed in a domain model in terms of features. FORM process is shown in Figure 3, where different types of features is considered: functions provided by products, technical operating environment, application domain technologies, and implementation techniques.

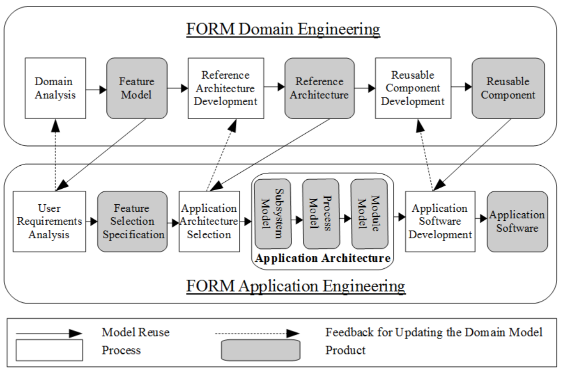

FORM Domain Engineering
Domain Analysis
Feature Model
Reference Architecture Development
Reference Architecture
Reusable Component Development
Reusable Component
User Requirements Analysis
Feature Selection Specification
Application Architecture Selection
Subsystem Model
Process Model
Module Model
Application Architecture
Application Software Development
Application Software
FORM Application Engineering
Model Reuse
Process
Feedback for Updating the Domain Model
Product

Figure 3. FORM Engineering Process

4.1 Feature location

By analyzing the product line reference architecture implementation and its 7 applications source code, configuration parameters and technical documentation including variability guide, we have identified the product line features. We divided the product line features into the two broad categories, one referring to reference architecture technical features and the other to application business logic features. The features represent functionality a user would select when customizing business applications. There are 25 server layer reference architecture technical features, and 26 client layer reference architecture technical features for application engineering process to select when composing an applications. Also, there are 49 business logic server features and 60 client presentation logic features to select when customizing an application. From these features we have selected 25 server reference architecture and 49 business logic server features for actual refactoring. The rest, mostly client presentation logic features, will be refactored once the new web-based reference architecture and its components are developed.
Figure 4 shows a partial feature diagram of server reference architecture implementation before refactoring.

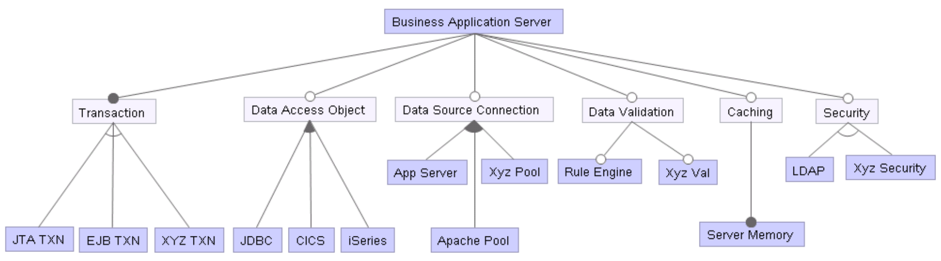


Figure 4. Feature model for reference architecture server subsystem

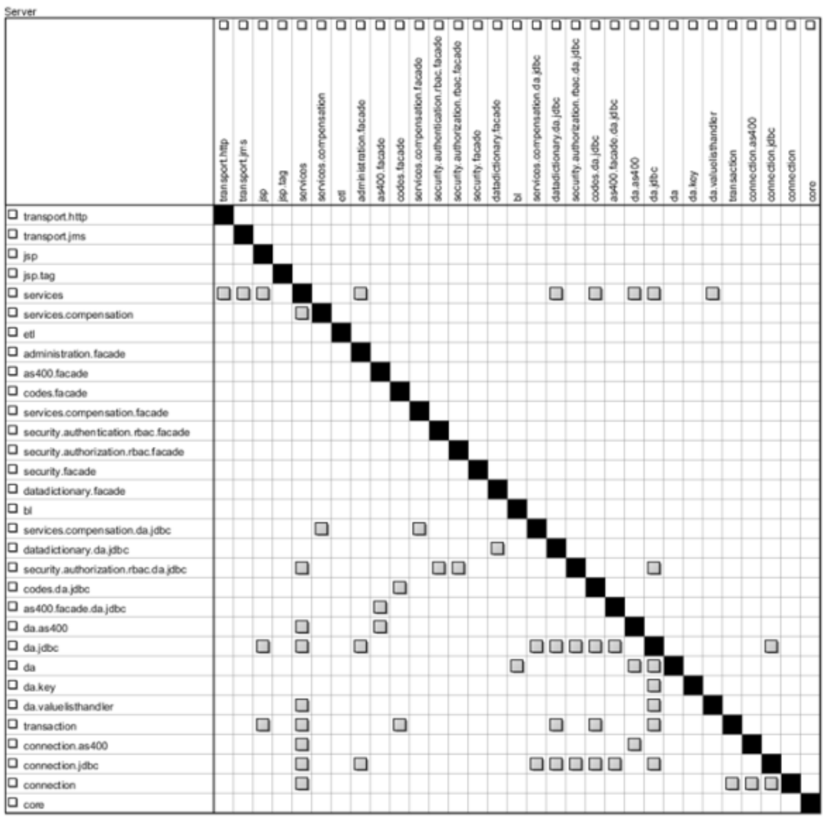


Figure 5. Initial components order

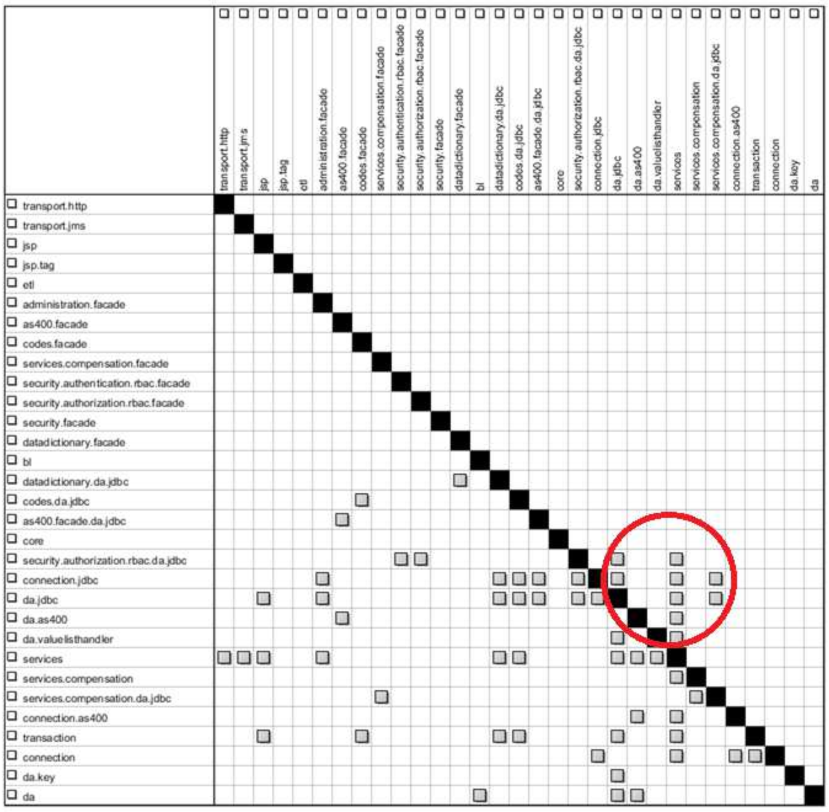
Table 1 shows the steps we have followed to do PLA analysis. After finding current state of product line features, we needed to analyze the Java source code to find the potential candidates for subsystem refactoring. To analyze the subsystem structural complexity we used Design Structure Matrix (DSM) tool named Cambridge Advance Modeler (2010). The method of analysis we applied is called sequencing. This form of partitioning analysis involves reordering rows and columns of the DSM to minimize cycles (i.e. to arrange the feature components with as many interactions as possible below the diagonal). The reordering of the DSM rows and columns is done in such a way that the new DSM transforming the DSM into an upper triangular form.

Table 1. Steps for analyzing SPL features for refactoring

Order	Activity
1	Analyze feature dependency clusters (DSM)
2	Find unused variability
3	Find unused features
4	Find fat products
5	Find duplicate code in alternative features
6	Analyze historical usage of PLA features
7	Analyze new user requirements

In this study we use reference architecture implementation feature component as a unit of analysis. Components as unit of analysis contain functions and data structures associated to the same feature. The initial order as shown at Figure 5 is the result of initial random analysis. However, after applying a *Partition Matrix* sequencing method the components are reordered as shown at Figure 6. The cells above the diagonal are marked (in circle), which means that circular dependency among the components exists.

Since we apply the layered architecture style we needed to focus on what to change to reach the goal of layered architecture. The results at Figure 6 suggests that a refactoring might be used in order to reduce the coupling among feature components.



	transport.http	transport.jms	jsp	jsp.tag	etl	administration.facade	as400.facade	codes.facade	services.compensation.facade	security.authentication.rbac.facade	security.authorization.rbac.facade	security.facade	datadictionary.facade	bl	datadictionary.da.jdbc	codes.da.jdbc	as400.facade.da.jdbc	core	security.authorization.rbac.da.jdbc	connection.jdbc	da.jdbc	da.as400	da.valuelisthandler	services	services.compensation	services.compensation.da.jdbc	connection.as400	transaction	connection	da.key	da
transport.http	■																														
transport.jms		■																													
jsp			■																												
jsp.tag				■																											
etl					■																										
administration.facade						■																									
as400.facade							■																								
codes.facade								■																							
services.compensation.facade									■																						
security.authentication.rbac.facade										■																					
security.authorization.rbac.facade											■																				
security.facade												■																			
datadictionary.facade													■																		
bl														■																	
datadictionary.da.jdbc													□		■																
codes.da.jdbc								□								■															
as400.facade.da.jdbc							□										■														
core																		■													
security.authorization.rbac.da.jdbc										□	□								■		□			□							
connection.jdbc						□									□	□	□		□	■	□			□		□					
da.jdbc			□			□									□	□	□		□	□	■			□		□					
da.as400							□															■		□							
da.valuelisthandler																					□		■	□							
services	□	□	□			□									□	□					□	□	□	■							
services.compensation																								□	■						
services.compensation.da.jdbc									□																□	■					
connection.as400																						□		□			■				
transaction			□					□							□	□					□			□				■			
connection																				□				□			□	□	■		
da.key																					□									■	
da														□							□	□									■

Figure 6. Sequenced components order

Software dependency can be static or dynamic. Static dependencies, generally known as “compile time” dependency use the concept that one component is required to compile another components. The tools used here are good enough to discover static dependencies, while dynamic dependencies such as ones that use Java reflections are possible to discover just by manual analysis.
PLA architecture documentation shows that dynamic dependencies are consistently used to identify and invoke server side business methods. In other cases the architecture documentation prefers to avoid using reflection.

4.2 Reference Architecture Analysis

Next, we have applied general refactoring steps, shown in Table 2. After applying these steps for general refactoring, we have found one external

Table 2. Steps for general refactoring analysis

Order	Activity
1	Find current status for external components
2	Find deprecated Java code in PLA components
3	Find warning and error Java code in PLA components
4	Find dead code (static) in PLA components
5	Find similar code in PLA components
6	Find large classes
7	Find bugs
8	Find unused classes
9	Find potential cross-cutting (aspects)

component that is not supported by supplier after Java 1.5 version and need to be replaced by an alternative solution. Eclipse IDE shows many Java classes, interfaces and methods used by current product line implementation that are deprecated. The alternatives have to be analyzed and sample have to be tested in order to apply them to the product line. All errors have to be corrected and warnings need to be suppressed or corrected by more appropriate programming instructions. PLA components with similar code are identified using Eclipse plugin CodePro Analytix^{TM} , tool [10]. We have identified 60 matches, a potential candidates to merge into one class. To find bugs we used *Eclipse Refactoring* feature and found 51 matches which needs to be corrected. We have found 4 potential candidate aspects to isolate in the reference architecture modules to be responsible for crosscutting concerns: exception handling, logging, performance measuring and business feature interface authorization. To facilitate communication among developers, refactoring analysis findings have been collected in refactoring catalog and describes using a uniform structure: name of activity, motivation, addressed code smell, actions needed for actual change, preconditions, priority, dependency, risk assessment, and etc.

5 Conclusion and future work

Business applications, in the context of software product lines, rely on a common reference architecture which is usually developed based on a client-server and layered architectural styles. The reference architecture is designed to provide coherent picture of the different components to be used throughout the different products. The components can be arranged into a useful configuration by restricting what each one is allowed to use. In this paper we use the terms subsystem to decompose of the whole product line into: reference architecture components, client components and server components.

The major objective of this study was to investigate the product line components relation types based on the implementation of software product line for business applications in a financial institution.

Investigating components refactoring in the context of product lines is interesting, because components are used not only in a single product, but they may be used to generate a whole family of products. Typically, refactoring is performed as a reaction to a code smells such as: duplicated code, long method, large class, long parameter list, however, software product lines give rise to a new group of code smells.

We plan to use the findings from this process as an input to the next step of converting current release of the product line PLA to the new refactored baseline.

6 References

[1] P. Clements and L. Northrop, *Software product lines: Practices and Patterns*. Addison-Wesley Boston, 2002.

[2] G. Booch, “Handbook of software architecture,” 2005.

[3] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, “FORM: A feature-; oriented reuse method with domain-; specific reference architectures,” *Ann. Softw. Eng.*, vol. 5, no. 1, pp. 143–168, 1998.

[4] W. F. Opdyke, “Refactoring object-oriented frameworks,” University of Illinois, 1992.

[5] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[6] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena, “Refactoring product lines,” in *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006, pp. 201–210.

[7] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, “FeatureIDE: A tool framework for feature-oriented software development,” in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, 2009, pp. 611–614.

[8] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[9] C. W. Krueger, “New methods in software product line development,” in *Software Product Line Conference, 2006 10th International*, 2006, pp. 95–99.

[10] *CodePro Analytix*. Google, Inc, 2011.