CrossFIRE: An Infrastructure for Storing Crosscutting Framework Families and Supporting their Model-Based Reuse

Rafael S. Durelli[1], Márcio E. Delamaro[1] and Valter V. de Camargo[2]

[1]Computer Systems Department University of São Paulo
São Carlos, SP, Brazil.

[2]Computing Departament
Federal University of São Carlos (UFSCAR)
São Carlos, SP, Brazil.

{rdurelli, delamaro}@icmc.usp.br[1], valter@dc.ufscar.br[2]

Company runs systems that have been implemented a long time ago. These systems usually are still under adaptation and maintenance to address current needs. Very often, adapting legacy software systems to new requirements needs to make use of new technological advances. Furthermore, legacy systems mainly consist of two kinds of artifacts: source code and databases. Usually, the maintenance of those artifacts is carried out through restructuring processes in isolated manners. Nevertheless, for a more effective maintenance of the whole system both should be analyzed and evolved jointly. Therefore, the lifespan of the legacy software systems are expected to improve. In this paper we put forward the MTBKDM, which is an infrastructure to assist the modernization of a legacy system.This infrastructure analyses Structured Query Language (SQL) queries embedded in a legacy source code in order to restructure and re-organize the system by using design patterns, such as Data Access Object (DAO) and Service-Oriented Architecture (SOA). In order to validate our approach we have carried out an experiment throughout a real-life case study. The results were promising regarding the effort employed to modernize a legacy software system.

1. Introduction

Companies have a vast number of operational legacy systems and these systems are not immune to software aging problems. However, they can not be discarded because they incorporate a lot of embodied knowledge due to years of maintenance and this constitutes a significant corporate asset. Furthermore, these kind of systems mainly consist of two artifacts: source code and databases. Usually, the maintenance of those artifacts is carried out through restructuring processes in isolated manners [**?**]. Nevertheless, we claim that for a more effective maintenance of the whole system both should be analyzed and evolved jointly.

Refactoring can be used in oder to deal with the maintenance of these systems. Martin Fowler [**?**] defines refactoring as "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". In the literature there is a lot of studies that make refactoring at the level of source code. Consequently, nowadays almost all Integrated Development Environmentd (IDE) provide some kind of automated support for program refactoring. Empirical studies of refactoring have shown that it can improve maintainability [**?**] and reusability [**?**] of legacy system. Not only does existing work suggest that refactoring is useful, but it

also suggests that refactoring is a frequent practice [**?**]. Cherubini and colleagues‘ survey indicates that developers rate the importance of refactoring as equal to or greater than that of understanding code and producing documentation [**?**].

Recent researches have been shifted from the typical refactoring process to the so-called Architecture-Driven Modernization (ADM) [**?**]. ADM has been proposed by OMG (Object Management Group). with the concept of modernizing legacy systems with a focus on all aspects of the current system architecture and the ability to transform current architectures to target architectures. In addition, ADM advocates conducting reengineering processes following the principles of Model-Driven Development (MDD) [**?**], i.e., it treats all the software artifacts involved in the legacy system as models and can establish transformations among them. Firstly a reverse engineering is performed starting from the source code and a model instance (**P**lataform **S**pecific **M**odel - PSM) is created. Next successive refinements (transformations) are applied to this model up to reach a good abstraction level (PSM) in model called KDM (**K**nowledge **D**iscovery **M**etamodel). Upon this model, several refactorings, optimizations and modifications can be performed in order to solve problems found in the legacy system. Secondly a forward engineering is carried out and the source code of the modernized target system is generated again. According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a multipurpose standard metamodel that represents all aspects of the existing IT (Information Technology) architectures. The idea behind the standard KDM is that the community starts to create parsers from different languages to KDM. As a result everything that takes KDM as input can be considered platform and language-independent.

Accordingly, one of the challenges of Software Engineering focuses on specifications, architectures and mechanisms to support the automation of software reengineering process, aiming to reduce time and effort spent in this process. Furthermore, legacy systems, often with high maintenance costs due to scarcity or absence of documentation, may be restructured and ported to ADM, providing greater quality in development and maintenance of these systems. Motivated by these ideas, we put forward an infrastructure based on the metamodel KDM which support legacy system reengineering from databases queries and source code, with the objectives of reducing the time and effort in this process and providing migration of these systems to ADM. More specifically, this infrastructure analyses Structured Query Language (SQL) queries embedded in a legacy source code in order to restructure and re-organize the system by using design patterns, such as Data Access Object (DAO) and Service-Oriented Architecture (SOA). This paper is organized as followed: Section 2 provides information related to the infrastructure - Section 3 the architecture of the infrastructure is depicted - in Section 4 there are related works and in Section 5 we conclude the paper with some remarks and future directions.

2. NOME DA INSFRA

In this section our infrastructure is presented. The theory behind it is based on the horseshoe modernization model, which is depicts in Figure 1. As can be seen, this model consist of three steps, **Reverse Engineering**, (*ii*) **Restructuring**, and **Forward Engineering**. The first step is represented by the left side of the horseshoe, it analyzes the legacy system in order to identify the components of the system and their interrelationships. Usually the reverse engineering step builds one or more representations of the legacy system at a higher level of abstraction (PSM). The second one is represented by the curve of the

horseshoe since this steps takes the previous system's representation (PSM) and transforms it into another one (PIM) at the same abstraction level. Finally, the last step is represented by the right side of the horseshoe because it generates physical implementations (source-code) of the target system at a low abstraction level from the previously restructured representation of the system.

It worth noticing that KDM is the most essential part of our infrastructure. KDM supplies the representation and management of knowledge extracted by means of reverse engineering from all the different software artifacts of the legacy system. Therefore, the legacy knowledge obtained is then modernized into a target improved system by using the concept of MDD. As the infrastructure herein follows the three steps depicts in Figure 1 then it has to go through five abstraction levels with four transformations among them.

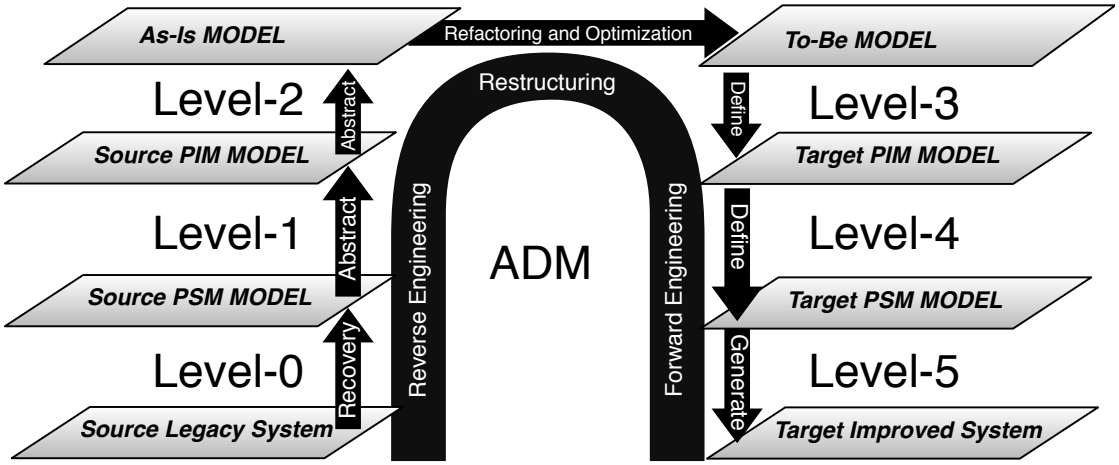

Figure 1. Horseshoe modernization model.

In this section the Crosscutting Framework Integrated Reuse Environment (CrossFIRE) is presented. Figure 2 depicts a screenshot of CrossFIRE.

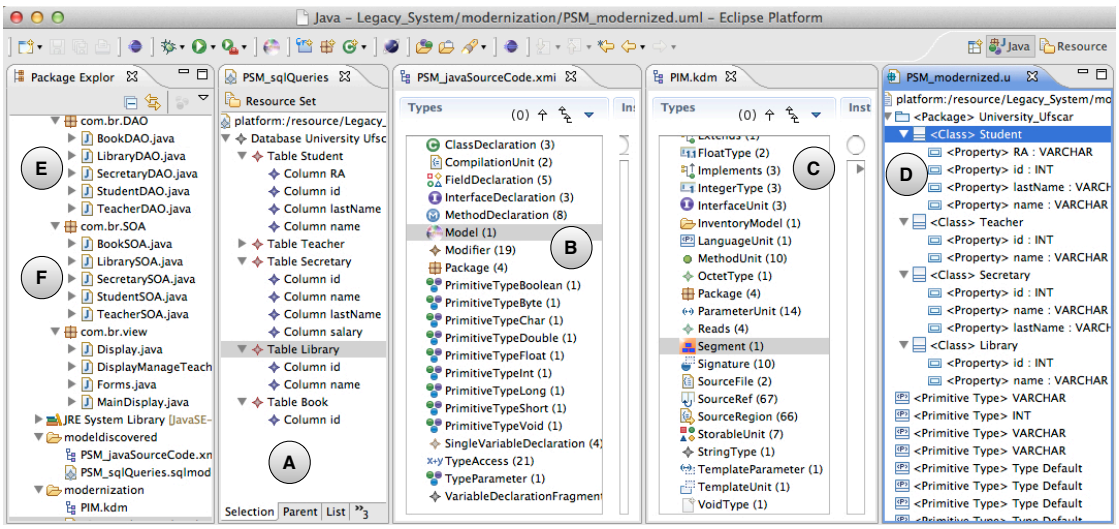

Figure 2. Screenshot of CrossFIRE

All artifacts of the CF/CFF must be developed before one uses the CrossFIRE. Thus, it is worth highlighting that CrossFIRE only supports the Application Engineering (AE) phase. Having this in mind, the domain engineer has to use a traditional approach in order to create the artifacts in the Domain Engineering (DE) phase. In the DE phase, three artifacts must be created: (*i*) source code of the whole CF, (*ii*) the RRM and (*iii*) the feature models of the CFF. These feature models have to be devised using the FeatureIDE[1], which is a tool that provides a graphical way to assemble feature models. Afterwards, the engineer can use the CrossFIRE in order to uploaded these artifacts into a repository, as is shown in Figure 2(A).

In the AE phase, an application engineer can browse the repository in order to check whether there are CFs that can be reused in order to implement concerns of the planned application. In order to assist this activity, CrossFIRE provides a visualization, which depicts all CFs/CFFs that have been uploaded by the domain engineers. Figure 2(B) shows the visualization of all CFs/CFFs available. As can be seen, there are six different CFFs - persistence, security, distribution, concurrency, logging and Business, respectively. In addition, CrossFIRE also shows descriptions for each of the selected CFFs by clicking on the button "Description". Nevertheless, if this description is not enough to help the engineer takes a decision on reusing the CF/CFF, CrossFIRE supplies a way to visualize the feature model related to selected CF/CFF by clicking on the button "View", Figure 2(C). As is shown in Figure 2(B), the "persistence" CF is highlighted to indicate that it has been chosen. Next, the "Download" button has to be clicked to transfer the feature model belonging to the CF chosen from the remote repository to the computer of the application engineer.

Furthermore, to reuse the CFF, its features must be chosen by the engineer aiming at specifying explicitly which features will be reused in the base application. To assist this activity, CrossFIRE uses the "configuration file" of FeatureIDE. By using this "configuration file", features can be chosen by the application engineer. The graphical notation of the "configuration file" is shown in Figure 2(D), which represents all features of the "persistence" CF. Moreover, FeatureIDE validates if the selected features match a valid combination for the instantiation of a member of the CFF. As shown in Figure 2(D), once the application engineer has chosen the features (represented by "+"), the resulting variant and constraints are generated automatically (represented by "-").

The application engineer should provide the selected features by using the "configuration file" to repository server, which will carry out an algorithm. This algorithm aims to extract two artifacts. The first extracted artifact only contains code related to the selected features. The second extracted artifact is a RM derived from a RRM, by removing unrelated requirements. The RM is used to support the reuse process of a CF. After that, these artifacts are sent to the application engineer's computer.

To reuse a member of the CFF or any other CF persisted in the repository, the application engineer may use the RM. It is graphically represented as a form which contains fields that should be filled with information regarding the base application. By completing this form, the code needed to couple the CF to the base application can be generated. It is possible to see our model editor in Figure 3. The "Palette" on the right of the figure

[1] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`

contains the elements of the RMs. They are: “Group”: an element to group any element visible in the models; “Pointcut”: represents join-points of the base application; “TypeExtension”: represents types found in the base application; “Value”: represents any numeric or textual values that must be informed while reusing the CF; “Option”: defines a selectable variability of the framework and “OptionGroup”: group selectable variabilities of the framework.

Each box contains a name and a description for the required information. The last line should be filled to provide the information regarding the base application. Note that the last line is only used in RMs. For example, to be able to instantiate a persistence CF, the application engineer must specify methods from base application that should be executed after a database connection is opened and before it is closed. Then, the box “Connection Opening” in Figure 3 represents names of methods that need an open database connection. It is also needed to specify methods that represent data base transactions, and the variabilities must be chosen, e.g., the driver which should be used to connect to the database system. After filling the fields of the RM, a model transformation generates the code needed to couple the CF, and then, the reuse process is complete. Our tool also provides validation of the information filled into RM models. At the moment this paper was written, only AspectJ CFs are supported.

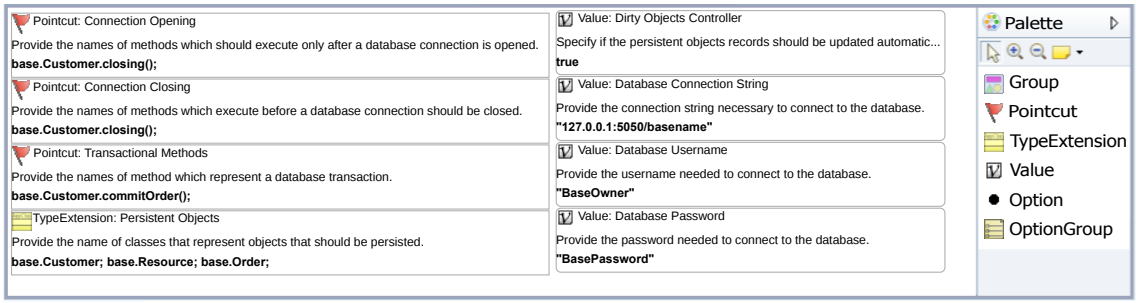

Figure 3. Reuse Model Editor

3. Architecture

In Figure 4 is depicted the architecture of the CrossFIRE. As shown in this figure, we devised the CrossFIRE on the top of the Eclipse Platform. On the other hand, both the RRM and RM have been developed using Eclipse Modeling Framework and Graphical Modeling Framework[2]. Moreover, we have studied FeatureIDE’s source code and extended it in order to use for CF/CFF, its feature model editor and its configuration file, which is used to extract member of a CFF.

As can be seen in Figure 4, all artifacts (source code, feature model, RM and RRM) that the CrossFIRE provides are persisted in a database. These artifacts are persisted in a remote server, available to be reused in the AE phase. This remote server is a physical computer, which is dedicated to run the RESTful API. Therefore, to send these artifacts by the server we have used this API as web service to cache the representation of all artifacts. This server receives requests of the CrossFIRE through RESTful, processes database queries and sends a response to the CrossFIRE by using RESTful as well. Furthermore, we have used Java Persistence API (JPA) 2.0 to deal with the way relational

[2]http://www.eclipse.org/modeling/gmp/

data is mapped to Java objects. To implement the database of the server, the MySQL was chosen.

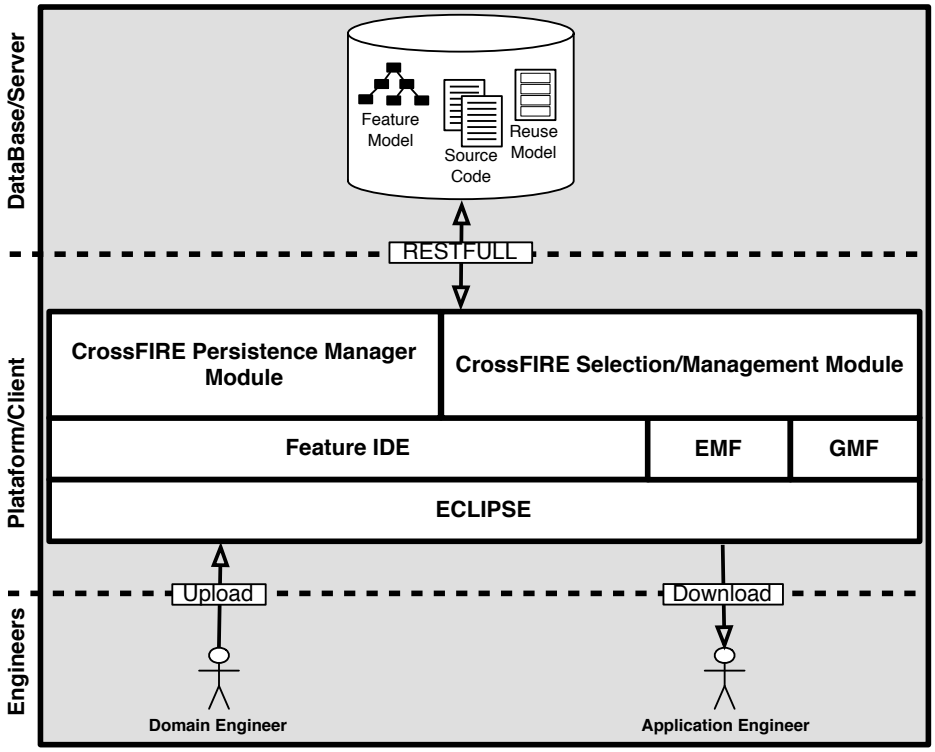

Figure 4. Architecture of CrossFIRE

4. Related Work

The approach proposed by Cechticky *et al.* [**?**] allows object-oriented application framework reuse by using a tool called OBS Instantiation Environment. That tool supports graphical models do define the settings of the expected application to be generated. The model to code transformation generates a new application that reuses the framework. The proposal found in this paper differs from their approach on the following topics: (*i*) their approach is restricted to frameworks known during the development of the tool; (*ii*) it does not use aspect orientation; (*iii*) the reuse process is applied on application frameworks, which are used to create new applications.

Another approach was proposed by Oliveira *et al.* [**?**]. Their approach can be applied to a greater number of object oriented frameworks. After the framework development, the framework developer may use the approach to ease the reuse by writing the cookbook in a formal language known as Reuse Definition Language (RDL) which also can be used to generate the source code. This process allows to select the variabilities and resources during reuse, as long as the framework engineer specifies the RDL code correctly.

5. Concluding Remarks

In the DE phase, CrossFIRE provides an infrastructure in which all of the CFF artifacts is developed. Afterwards, the CrossFIRE allows the engineers share these artifacts. As a result, these artifacts will be stored in a remote repository to be reused in the AE phase.

In the AE phase the CrossFIRE shows a list of all available CFFs that have been shared. Therefore, the engineer can pick out which CFF(s) can be reused in his base application. Next, the CrossFIRE provides a way to perform the download of the feature models related to each CFF. Through these feature models the engineer can pick out which features his base application really requires. As a consequence, the CrossFIRE downloads two artifacts, the necessaries chunks of code and a RM. Thus, the application engineer fills in the RM with the information needed by an member of a CFF's, and after that, it is possible to generate the final reuse code.

We believe that CrossFIRE increases the degree of reuse and allows the engineer to avoid dead code in his base application. Moreover, this infrastructure aims make the reuse activities easier. Long term future work involves conducting experiments to evaluate the level of reuse provided by CrossFIRE. It is worth highlighting that CrossFIRE is open source and it can be downloaded at *www.dc.ufscar.br/∼valter/crossfire*.

6. Acknowledgements

The authors would like to thank CNPq for Processes 132996/2010-3, 560241/2010-0 and 483106/2009-7 and FAPESP for Process 2011/04064-8.

References

Cechticky, V., Chevalley, P., Pasetti, A., and Schaufelberger, W. (2003). A generative approach to framework instantiation. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 267–286, New York, NY, USA. Springer-Verlag New York, Inc.

Oliveira, T. C., Alencar, P., and Cowan, D. (2011). Reusetool-an extensible tool support for object-oriented framework reuse. *J. Syst. Softw.*, 84(12):2234–2252.