Using Refactoring and Unification Rules to Assist Framework Evolution

Mariela Cortes
Computer Science Department
Pontifical Catholic University of Rio de Janeiro
mariela@inf.puc-rio.br

Marcus Fontoura
IBM Almaden Reserch Center
650 Harry Road,
San Jose, CA, USA, 95120
fontoura@almaden.ibm.com

Carlos Lucena
Computer Science Department
Pontifical Catholic University of Rio de Janeiro
lucena@inf.puc-rio.br

ABSTRACT

Although object-oriented software development has experienced the benefits of using frameworks, a thorough understanding of how to change them to meet evolving requirement needs is still object of research. Therefore framework development is very expensive, not only because of the intrinsic difficulty related to capturing the domain theory, but also because of the lack of appropriate methods and techniques to support the evolution and redesign of the framework architecture. This paper proposes the use of refactoring and unification rules to assist framework evolution. The approach is illustrated through the JUnit testing framework.

KEY WORDS: object-oriented frameworks, framework redesign and evolution, refactoring, unification rules.

1. INTRODUCTION

Currently, there is very little support for framework evolution. Some of the most common problems regarding framework evolution include how to evolve its architecture without impacting in the applications previously created, how to incorporate new domain requirements in the design, and how to control the concurrent evolution of frameworks and applications [3].

In this paper we show how refactoring [8, 10, 12] and unification [6] rules can be used to assist framework maintenance and evolution. We show how to analyze the framework architecture to verify if a given adaptation is feasible or not. Based on this analysis, we show how refactoring and unification rules can be applied to make unfeasible adaptations possible. Refactorings are behavior-preserving transformations that improve the framework design. We are particularly interested in refactorings that make the design more flexible. Unifications are non-preserving transformations, especially useful to incorporate new features to a design. The combination of refactorings and unifications to evolve framework designs in a controlled way is the key point of this work.

The rest of this paper is organized as follows: Section 2 presents the problems related to framework evolution and how refactoring and unification rules can be used to address them. Section 3 illustrates how refactorings and unifications can be applied in practice, using the JUnit testing framework [2] as an example. Section 4 proposes a unification-based framework development process, which incorporates many of the ideas from the eXtreme Programming (XP) methodology [1]. Section 5 describes some related work. Finally, section 6 presents our conclusions and future research directions.

2. PROBLEMS WITH FRAMEWORK EVOLUTION

Some of the most common problems regarding framework evolution are described in [3] and summarized below:

- Structural complexity: framework evolution can make its structure (object interfaces, class hierarchies, and so on) hard to manage and understand;
- Changes in the domain: as the framework evolves and new framework instances are created, new abstractions that should be part of the framework may be derived;
- New design insights: the framework's design structure may need to be improved in the light of issues previously neglected or forgotten.

In all the three cases an explicit definition of how to evolve the framework architecture is required. Changing the implementation of an existing framework class every time new functionality is needed, for example, is not good practice. There are two complementary approaches to address the evolution issues*:* refactoring [8, 10, 12] and unification [6] rules. Refactorings are behavior-preserving transformations that may be applied to the design and implementation artifacts. Examples of refactorings are moving common behavior to super classes, renaming methods, and so on.

An example of a design-level refactoring is the division of a method into two complementary methods. As an example, method *taxReport()* is decomposed into *getExpenses()* and *calculateTax()*, as shown in Figure 1. This is a valid design refactoring if the semantics of the method is preserved. This refactoring can be useful, for example, if the *getExpenses()* method is used in other places in the same system.

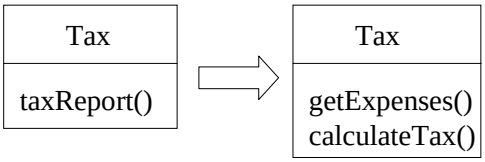

Figure 1. Splitting method refactoring

Unification rules define a way to incorporate new abstractions into the framework structure. As in refactoring, unification rules are transformations on the framework design structure. Unlike refactoring, unification rules are not behavior-preserving: the semantics of the framework is changed to incorporate the semantics of the new features.

Considering the design presented in Figure 1, a unification situation exists if variations of the *taxReport()* implementation are required by a given framework instance. Since *taxReport()* is not a variation point, the instance would have to violate the framework architecture to redefine its behavior.

Figure 2 illustrates a unification procedure used to avoid this problem, in which the domain changes represents the new requirements where the *taxReport()* method semantics differs from the one previously supported by the framework.

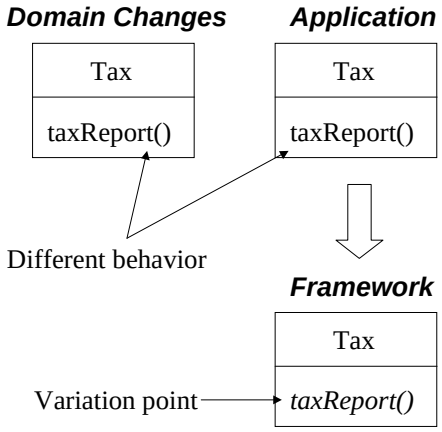

Figure 2. Unification rule example

Sometimes the application of refactoring procedures before the unification can improve its result. In this same example, it might be the case that the *taxReport()* method varies only when calculating the taxes content but is stable when getting the expenses. Then, if the splitting refactoring had been applied before the unification the variation point would be *calculateTax()* only, and not the entire *taxReport()* method.

Framework adaptation normally takes place by completing the variation points defined by its architecture. However, there are many cases in which the framework does not support the required customization and the application developers need to violate its structure. This phenomenon is referred to as architectural drift: the intended framework architecture and the architecture that underlies the current implementation of a given framework instance become different [3]. Unification rules can be used to avoid this phenomenon, by making the necessary transformations in the framework structure to incorporate the changes required by a given framework instance. The term *unification* is used to indicate that the rules are used to "unify" the framework architecture with the architecture of the violating instance.

Currently there are three unification rules, which are defined based on the basic frameworks patterns proposed by Pree [13]. Framework patterns define the possible ways of implementing variation points as a combination of template and hook methods. There are three different kinds of framework patterns:

- Unification, in which the variation point (hook) and the method that invokes it (template) belong to the same

class. The Template Method and Factory Method design patterns [9] are based on this basic framework pattern.

- Separation, in which the template and hook methods belong to different classes. There are several patterns from [9] based on this principle, including State, Strategy, Command, and Bridge.
- Recursive patterns, in which the template and hook methods are arranged in a recursive object structure, such as in the Composite, Decorator, and Chain-of-Responsibility patterns [9].

Therefore, currently there are three unification rules defined "Add unification pattern", "Add separation pattern", and "Add recursive pattern". There are other possible unification rules that can be defined for variation points that are not implemented as template-hook combinations, e.g. based on reflection, C++ templates, and so on.

3. THE JUNIT CASE-STUDY

To illustrate the proposed approach, we will use the JUnit testing framework [2] as an example. We chose this framework because it is simple enough to be described in a paper, yet it is functional, allowing the automation of unit tests of Java components. The evolution of the JUnit framework is represented in terms of diagrams that show snapshots of its architecture. The notation used here is the same proposed by its authors [2]. It extends UML class diagrams [11, 19] with annotations that indicate the design pattern [9] used for each of the framework components. The code presented in this section was taken from [2].

3.1 FIRST ITERATION: DEFINITION OF TEST CASES

The framework is designed to address the problem of testing Java components. Its initial architecture is the simplest one that allows for the definition of automated test cases. Each test case determines whether a particular aspect is carried out correctly. A test case is defined in JUnit by creating an instance of *TestCase*, and overwriting the abstract method *run()*, as shown in Figure 3.

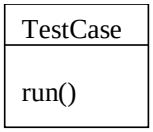

Figure 3. First iteration

Figure 4 illustrates an adaptation, through a UML collaboration that follows the Catalisys approach [4]. It represents the JUnit framework, and the application class *TestEmployee* (e.g. used to test a given *Employee* class). This diagram indicates that *TestEmployee* plays the role [14, 15] of *TestCase* in the adaptation.

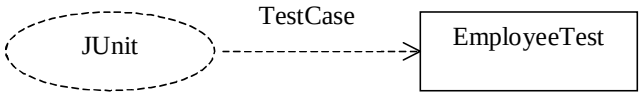

Figure 4. First adaptation example

3.2 SECOND ITERATION: REUSING TEST CASE CODE

The problem of the first version of the framework is that it does not allow for any code reuse among related test cases. However, there exist similar steps that are executed in the same order for any test cases, although their implementations may vary. These steps are: set up, in which the test variables are defined, test execution, in which the test verifications occur, and clean up, in which any recourses used during the testing may be released, such as opened files. The idea is to evolve the framework architecture to take profit of this organization of test cases, and therefore allow code reuse, for instance allowing to test case to use the same set up code.

This evolution is possible through the application of the "Form template method" refactoring [8]. This refactoring allows us to decompose a method into several primitive operations while preserve the same behavior. After the refactoring is applied, the new structure for the JUnit framework is the one shown in Figure 5.

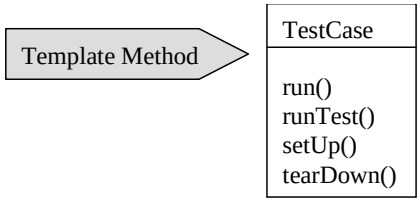

Figure 5. Second iteration

The evolution had an effect over the *run()* method, previously defined an abstract method:

```
public abstract void run();
```

Converting it into:

```
public void run(){
        setUp();
        runTest();
        tearDown();
}
```

The abstract methods *runTest(), setUp(),* and *tearDown()* are generated by the application of the refactoring. This refactoring has some effect on the previously created test cases, that now have to implement *setUp(), runTest(),* and *tearDown(),* instead of implementing *run().* Although the old approach of overriding the *run()* method still works, it does not allow for code reuse as described above.

Note that the refactoring was applied here since *run()* was already defined as a variation point. In the case that it was not, we would have to apply a unification rule to create the variation point. The most suitable rule for this situation would be "Add unification pattern", since the template and hook methods belong to the same class.

3.3 THIRD ITERATION: SHOWING TEST RESULTS

Another evolution step is to make reporting of testing results flexible. In the current solution the code that does the result reporting is mixed with the testing code. This can be avoided by the creation of a new entity, *TestResult,* which would carry the error reporting.

This step of evolution is a non behavior-preserving transformation. Consequently there are no refactorings that can be applied and the unification rules come into the game. In this case, since we want the *TestResult* entity to be decoupled from the *TestCase,* the "Add separation pattern" is the best option. Figure 6 illustrates the new design after the unification is performed. Figure 7 illustrates a new collaboration diagram for JUnit, which highlights that now two bindings are required to adapt the framework: *TestCase* and *TestResult.*

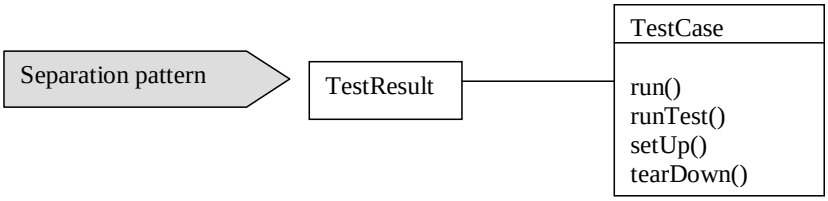

Figure 6. Third iteration

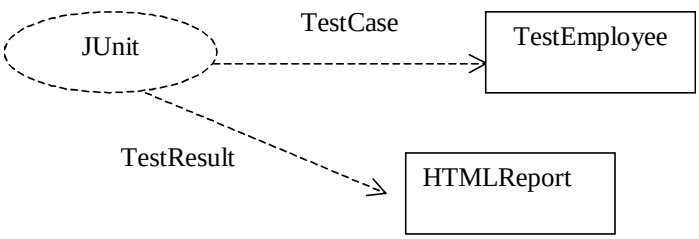

Figure 7. New JUnit collaboration diagram

3.4 FOUTH ITERATION: CREATING TEST SUITES

A last improvement considered for the JUnit evolution is to extend it so that it can run simultaneously many different tests. In this situation, the most *run()* test is a recursive variation point, which can be applied to individual test cases as well as to test suite. This situation is shaped in Figure 8 using the Composite design pattern [9].

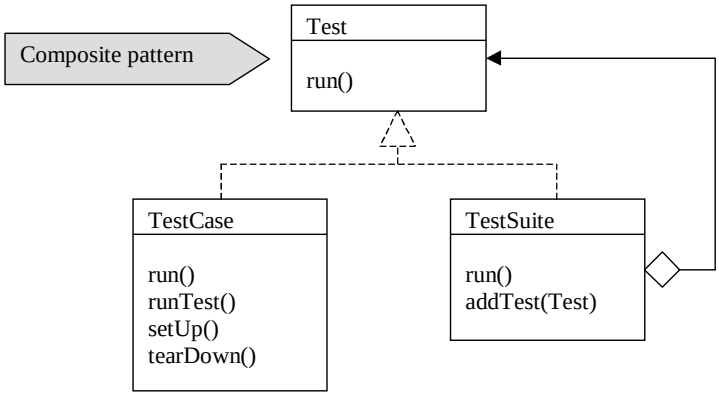

Figure 8. Flexibilization for running suites of tests

Although flexible, the definition of test suites is a black-box variation point, since the customization is made without the addition of new classes, but by the definition of the list of objects that compose the suite. Thus, the application of the "Add recursive pattern" unification has no effect over the collaboration diagram shown in Figure 7.

4. A UNIFICATION-BASED FRAMEWORK DEVELOPMENT PROCESS

There are various application areas that are not yet established and for which new ideas and models are presently under development and evaluation. These are domains in which the possibility of rapidly building new applications is essential and strategic from a practical point of view. Examples of application domains that can be classified in this category include web-based education, electronic commerce, computational biology, and finances.

It is very difficult to develop an adequate framework for these domains without a lot of experimentation. One possible solution for automating part of the job is the use of a unification-based development process.

The idea is to develop a first approximation of the framework and start to develop applications from it. Some of these applications will violate the framework architecture. Every time this happens, the unification rules may be applied generating a more mature version of the framework. When no more unifications are required the framework can be considered sufficiently mature. Figure 9 illustrates this process.

The ALADIN framework [7] was defined using this approach, as described in [6]. However, until now this was our only experience with a framework development based on unification rules. In order to evaluate the merits and the limitations of the approach new case studies need to be developed.

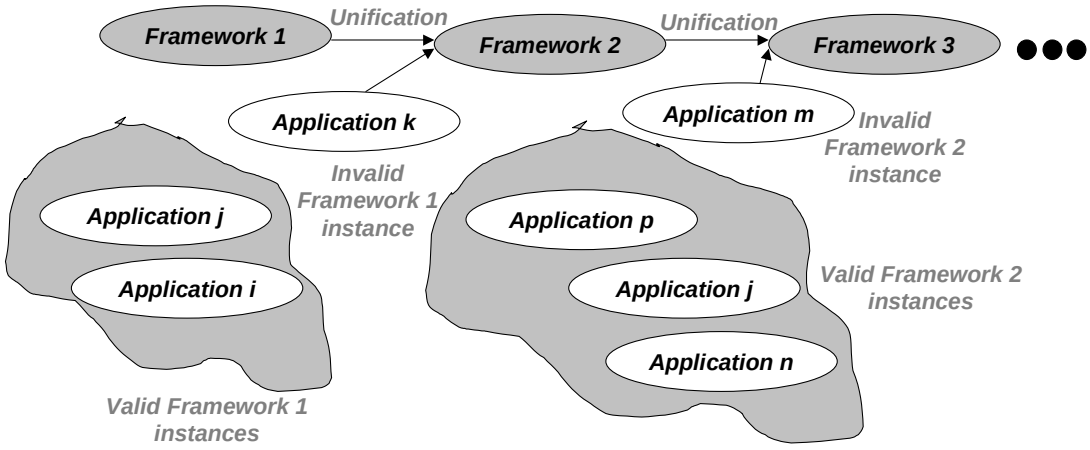

Figure 9. Unification-based software development

This process follows the ideas in the eXtreme Programming (XP) methodology, which advocates that software should be developed only the minimal required functionality, and that new functionality should be added only when needed. The unification-based process applies this approach to framework development: you should build the simplest framework that solves the domain problem. If a given adaptation is not possible, then modify the framework to make it feasible. Unification rules assist you do that.

5. RELATED WORK

Currently there are very few framework design methods that deal with framework evolution. A pattern-based description of some accepted approaches underlying framework design can be found in [17]. Some interesting aspects regarding framework design such as framework integration, version control and over-featuring can be found in [3].

The Refactoring Browser [16] is a tool to help maintenance of frameworks written in Smalltalk. It currently does not support unification rules, but it has an open architecture and the introduction of unification and new refactoring procedures seems to be straightforward. The design pattern tool proposed in [5] also uses refactorings to achieve framework restructuring.

Roberts and Johnson propose the development of concrete applications before actually developing the framework itself [17]. They claim that framework abstractions can be derived from concrete applications. The unification-based development process may be used to systematize this approach. An approach that integrates framework and XP is presented in [18].

A model for framework development based on viewpoints is proposed in [6]. This method was used as our first approach to framework design, and the current version has been refined through the development of several case studies.

6. CONCLUSIONS AND FUTURE WORK

This paper shows how unification rules can be combined to refactoring rules to support framework maintenance and evolution. Unifications are transformations used to avoid the architectural drift problem [3] by restructuring the framework hot-spots during evolution. After the application of a unification transformation the set of applications that may be instantiated based on the framework is enlarged, since new variation points are defined. Unification rules can be used as a basis for a development process that produces frameworks for "changing" domains. This idea can be seen as a systematization of the generalization from concrete examples approach proposed in [17]. We currently plan to add tool support for unification rules and to consider other rules for different kinds of variation points.

REFERENCES

1. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
2. K. Beck and E. Gamma, "JUnit: A Cook's Tour" (http://www.junit.org), 1998.
3. W. Codenie, K. Hondt, P. Steyaert, and A. Vercammen, "From Custom Applications to Domain-Specific Frameworks", *Communications of the ACM*, 40(10), 71-77, 1997.
4. D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1999.
5. G. Florijin, M. Meijers, P. van Winsen, "Tool Support for Object-Oriented Patterns", ECOOP'97, *LNCS 1241*, Springer-Verlag, 472-495, 1997.
6. M. Fontoura, S. Crespo, C. Lucena, P. Alencar, and D. Cowan, "Using Viewpoints to Derive a Object-Oriented Frameworks: a Case Study in the Web-based Education Domain", Journal of Systems and Software, *Elsevier Science*, 54(3), 239-257, 2000

7. M. Fontoura, L. Moura, S. Crespo, and C. Lucena, “ALADIN: An Architecture for Learningware Applications Design and Instantiation”, Technical Report MCC34/98, Computer Science Department, PUC-Rio, 1998.

8. M. Fowler, *Refactoring: Improving the design of existing code*, Addison-Wesley, 1999.

9. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

10. R. Johnson and W. Opdyke, “Refactoring and aggregation”, Object Technologies for Advanced Software, First JSSST International Symposium, *LNCS 742*, 264-278, 1993.

11. OMG, “OMG Unified Modeling Language Specification V.1.2”, 1998 (http://www.rational.com/uml).

12. W. Opdyke, “Refactoring Object-Oriented Frameworks”, Ph.D. Dissertation, Computer Science Department, University of Illinois, Urbana-Champaign, 1992.

13. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

14. T. Reenskaug, P. Wold, and O. Lehne, *Working with objects*, Manning, 1996.

15. D. Riehle and T. Gross, “Role Model Based Framework Design and Integration”, OOPSLA’98, *ACM Press*, 117-133, 1998.

16. D. Roberts, J. Brant, and R. Johnson, “A Refactoring Tool for Smalltalk”, University of Illinois at Urbana-Champaign, Department of Computer Science (http://st-www.cs.uiuc.edu/users/droberts/).

17. D. Roberts and R. Johnson, “Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks”, in *Pattern Languages of Program Design 3*, Addison-Wesley, R. Martin, D. Riehle, and F. Buschmann (eds.), 471-486, 1997.

18. S. Roock, “eXtreme Frameworking - How to aim applications at evolving frameworks”, Proceedings of the XP’2000 Conference, 2000.

19. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.