Using UML-FW, Refactoring, and Unification Rules to Assist Framework Maintenance and Evolution

Marcus F. Fontoura
Department of Computer Science
Princeton University
35 Olden Street, Princeton, NJ
08544, U.S.A.
fontoura@cs.princeton.edu

Carlos José P. de Lucena
Computer Science Department,
Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente, 225
22453-900 Rio de Janeiro, Brazil
lucena@inf.puc-rio.br

ABSTRACT

Although object-oriented software development has experienced the benefits of using frameworks, a thorough understanding of how to identify, design, implement, and 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 framework design and evolution. This paper proposes the use of the UML-FW profile, together with refactoring and unification rules to assist framework maintenance and evolution.

KEY WORDS: object-oriented frameworks, framework design, UML-FW, refactoring, unification rules.

1. INTRODUCTION

A *framework* (also referred to as *Product Line Architecture)* is a collection of several fully or partially implemented components with largely predefined cooperation patterns between them. A framework implements the software architecture for a family of applications with similar characteristics [14], which are derived by specialization through application-specific code. Hence, some of the framework components are designed to be replaceable. We call these components the variation points or hot-spots [15] of the framework. An application based on such a framework not only reuses the frameworks source code, but more important, its architecture design. This amounts to a standardization of the application structure and allows a significant reduction of the size and complexity of the source code that has to be written by developers who adapt a framework.

Recent standardization efforts of the Unified Modeling Language (UML) [12, 22] offer a chance to harness UML as notational basis for framework development projects. UML is a multi-purpose language with many notational constructs and a built-in extension mechanism. However, the current standard UML does not provide adequate constructs to model frameworks. There is no indication in UML design diagrams what are the variation points and what are their instantiation constraints. Fortunately, UML provides extension mechanisms that allow us to define appropriate labels and markings for the UML model elements. Based on these mechanisms we have defined a new UML profile [4] called UML-FW especially useful for framework description, both during development and instantiation of a framework.

In [6, 9] we have shown how UML-FW can be used to automate framework development and instantiation. In this paper we focus on framework maintenance and evolution and show how refactoring [11, 13] and unification [7] rules can profit from an explicit representation of the framework hot-spots.

The rest of this paper is organized as follows: Section 2 describes UML-FW and how it can be applied to model a simple example in the web-based education domain. Section 3 presents the problems related to framework maintenance and evolution and how refactoring and unification rules can be used to address them, once frameworks are modeled with UML-FW. Section 4 proposes a unification-based framework development process. Section 5 describes some related work. Finally, section 6 presents our conclusions and future research directions.

2. UML-FW: MODELING FRAMEWORKS AT DEISGN LEVEL

Standard OO design methods, like OMT [21] and UML [12, 22], provide static (class and object diagrams) and dynamic (activity diagrams, interaction diagrams, state-chart diagrams, and state diagrams) views of a system design. Figure 1 is a representation of a student sub-system of a web-based education framework [8] in UML, where (a) shows a static view of the system (class diagram) and (b) shows a dynamic view (interaction diagram). In the dynamic view we show the interaction between an instance of each of the two classes.

The *showCourse()* method is the one responsible for controlling the application flow: it calls *selectCourse()*, which allows the student to select the desired course, *tipOfTheDay()*, which shows a start-up tip, and finally *showContent()* to present the content of the selected course.

The method *selectCourse()* is the one responsible for selecting the course the student wants to attend. It is a framework hot-spot since it can have different implementations in different web-based applications created within the framework (framework instances). Different examples of common selection courses mechanisms include: to require a student login or not, to show the entire list of available courses or just the ones related to the student major, and to show a course preview or not, and so on. There are an infinite number of possibilities that depend on the framework use.

Figure 1 shows *selectCourse()* as an abstract method of an abstract class, *SelectCourse*. The framework user has to create a subclass from *SelectCourse* and then provide a concrete implementation for *selectCourse()*. The problem with this representation is that there is no indication that this method is a hot-spot in the design diagrams. There is not also any indication of how the hot-spot should be instantiated. Although the name of the abstract method *selectCourse()* is italicized this notation is not an indication of a hot-spot, rather it indicates an abstract method; an abstract method does not necessarily have to be a hot-spot.

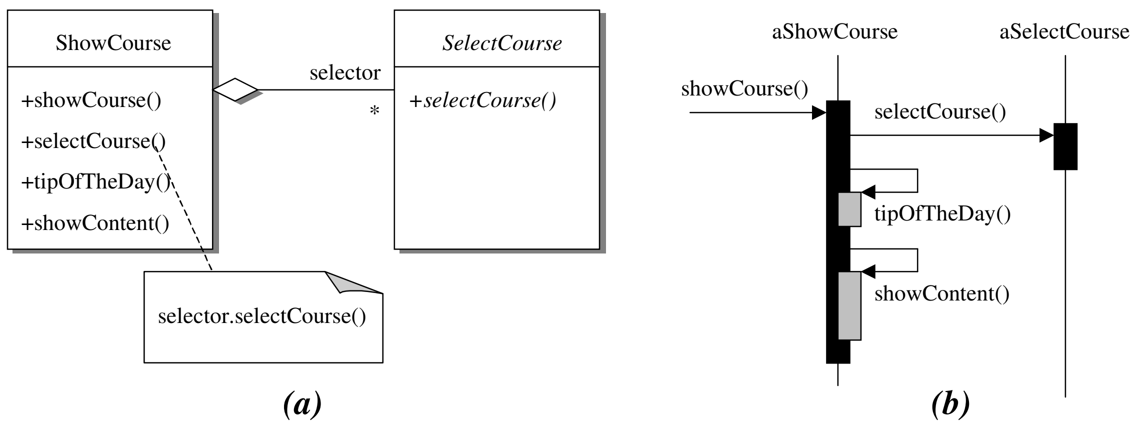

Figure 1. UML representation of a framework

The method *tipOfTheDay()* is also a framework hot-spot. The reason it that it may be wanted by some instances but may be unwanted by others. The framework should provide only the methods and information that are useful for all possible application instances in its kernel classes. The extra functionality, which may be referred to as should be provided only in the application code. A notation to designate this kind of hot-spots would be extremely useful in framework design. A design for the class *ShowCourse* could include a potentially infinite number of methods in the class, many of which would not be needed in a given application. Also, the inclusion of methods like *tipOfTheDay()* could result in a complex interface for *ShowCourse,* which is a bad design heuristic [18].

This simple example shows that standard OO modeling notations could be usefully augmented with constructors to distinguish hot-spots from the framework kernel, since there are several problems that are directly related to this "standard" design representation:

- Hot-spot modeling: the framework hot-spots are not first class citizens in the standard design languages. They have to be represented with standard object-oriented constructors, which has two undesirable consequences. The first is the difficulty of identifying the hot-spots in the framework structure, encumbering the instantiation process. The second is the complexity of the design diagrams, especially in the case that "tricks" are used add flexibility and extensibility to the framework, overcoming the lack of appropriate hot-spot constructors. Design patterns [1, 10, 23] are examples of "tricks" that might be used;
- Changing the implementation of an existing framework class every time new functionality is needed, as in the case of method *tipsOfTheDay()*, is also not good practice. If these hot-spot methods could be marked, the person who is creating the application from the framework and the application maintainer would know how they are invoked and which methods could be removed without disturbing the system.

There are basically three kind of hot-spots in object-oriented frameworks: *variable methods*, *extensible classes*, and *extensible interfaces*. Variable methods are methods that the implementation may vary, depending on the framework instantiation. In the example, method *selectCourse()*, is a variable method. Extensible classes are classes that the interface depends on the framework instantiation, like the *ShowCourse* class, which may require new instance methods *(tipsOfTheDay()* for example) depending on the instantiation. Extensible interfaces are UML interfaces (that can be viewed as a design model for Java interfaces) or abstract classes that require the creation of concrete (sub)classes to provide the desired functionality. The instantiation of this last kind of hot-spot is made through the creation new classes, which are called *Application Classes* since they exist only in the applications created from the framework.

In the web-based education framework [8] class *Actor* is an extensible interface hot-spot since new types of actors may be defined depending on a given framework instance. The default actor types are students, teachers, and administrators, however new types may be needed such as monitors, advisors, and secretaries. This design structure is presented in Figure 2.

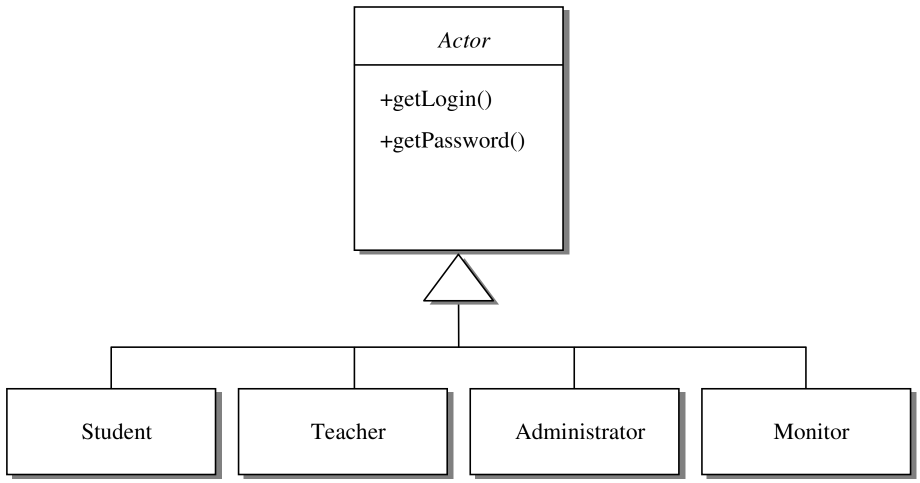

Figure 2. Example of an extension interface hot-spot

It should be clear that these three kinds of hot-spots have different semantics: in variable hot-spots the method implementation varies, in extensible classes the class interface varies, and in extensible interfaces the types in the system vary (new classes are required).

Our proposed solution for enhancing the framework representation extends UML by changing the class diagrams to explicitly identify framework hot-spots and instance classes. In fact these set of extensions define a new UML profile [4] for modeling frameworks, called UML-FW [6, 9].

To extent UML class diagrams a new stereotype was created, *Application Class*, and two new tagged values were defined, *variable* and *extensible*, to represent variable methods and extensible classes, respectively. Finally, the applicability of the *incomplete* constraint was modified to identify extension interfaces hot-spots. Table 1 summarizes these elements and defines their informal semantics.

Element	Type	Applies to	Informal Semantics
Application Class	Stereotype	Class	Classes that exist only in framework instances. New application classes may be defined during the framework instantiation.
Variable	Tagged value	Method	Means that the method implementation depends on the framework instantiation. Is used to identify *Variable Method* hot-spots.
Extensible	Tagged value	Method and Attribute Containers	Means that the class interface depends on the framework instantiation: new methods and attributes may be defined to extend the class functionality. This element identifies *Extensible Class* hot-spots.
Incomplete	Constraint	Generalization and Realization	Almost the same meaning that in standard UML. Incomplete means that new classes that satisfy a given relationship (generalization or realization) may be added during the framework instantiation, identifying *Extensible Interfaces* hot-spots.

Table 1. Summary of the new elements and their meaning

Figure 3 shows the web-based education framework class diagrams through the use of this extended notation. It represents the framework architecture in a more expressive way than if standard UML was used. The identification of the hot-spots by the keywords *extensible*, *variable*, and *incomplete* indicate what are and the exact meaning of each of the hot-spots. The application classes indicate what parts of the system have to be changed to create valid framework instances.

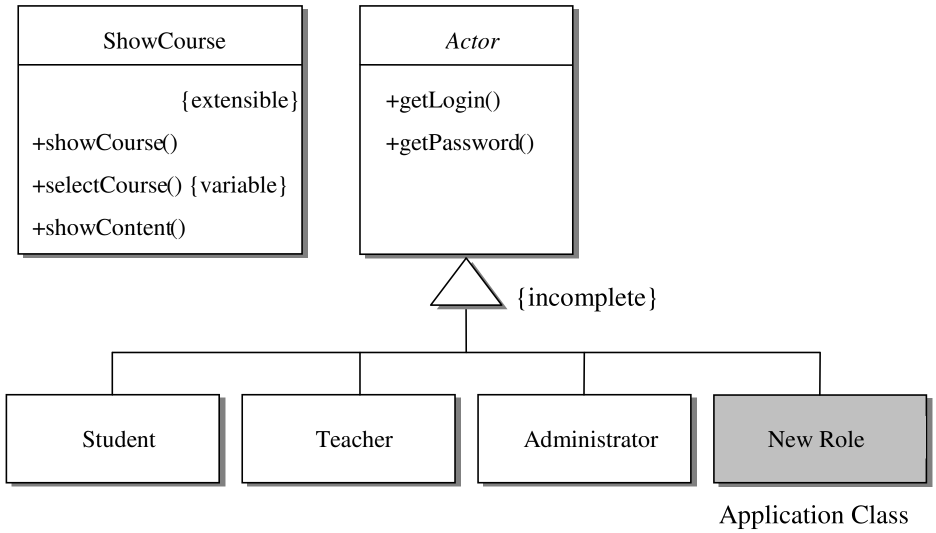

Figure 3. The UML-extended class diagram

The extended class diagrams completely specify the framework hot-spots and their semantics. It is very important that framework developers provide documentation that describes what parts of the system should be changed to create a valid framework instances. It is very unlikely that a framework user will be able to browse the framework code, which generally has complex and large class hierarchies, and write the appropriate code if the framework is not well documented. These new UML extensions address this problem.

3. FRAMEWORK MAINTENAINCE AND EVOLUTION

Currently, there is very little tool support for framework maintenance and evolution. Some of the most common problems regarding framework evolution are described in [2] 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 the framework's hot-spots may facilitate the job of the framework's maintainer. Changing the implementation of an existing framework kernel class every time new functionality is needed, for example, is not good practice. If the hot-spots were appropriately marked, both framework and application maintainers would know how they relate to the rest of the system and which methods and classes can be removed without disturbing the framework or the instantiated applications.

There are two complementary approaches to address the maintenance and evolution issues*:* refactoring [11, 13] and unification [7] 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 applied to the design represented in Figure 3 is the division of the *showContent()* method into two complementary methods: *getContent()* and *displayContent(),* as shown in Figure 4. This is a valid design refactoring if the semantics of the method is preserved. This refactoring can be useful, for example, if the *getContent()* method is used in other places in the same system.

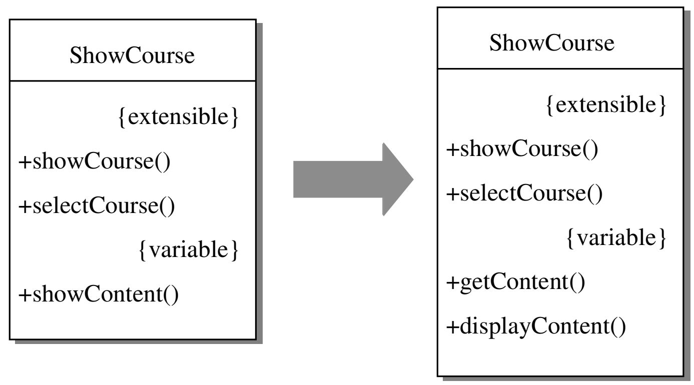

Figure 4. Splitting method refactoring

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 3, a unification situation exists if variations of the *showContent()* implementation are required by a given framework instance. Since *showContent()* is not part of the framework hot-spot structure, the instance would have to violate the framework architecture to redefine *showContent()*.

Figure 5 illustrates a unification procedure used to avoid this problem, in which the domain changes reprtesents the new requirements where the *showContent()* method semantics differs from the one previously supported by the framework.

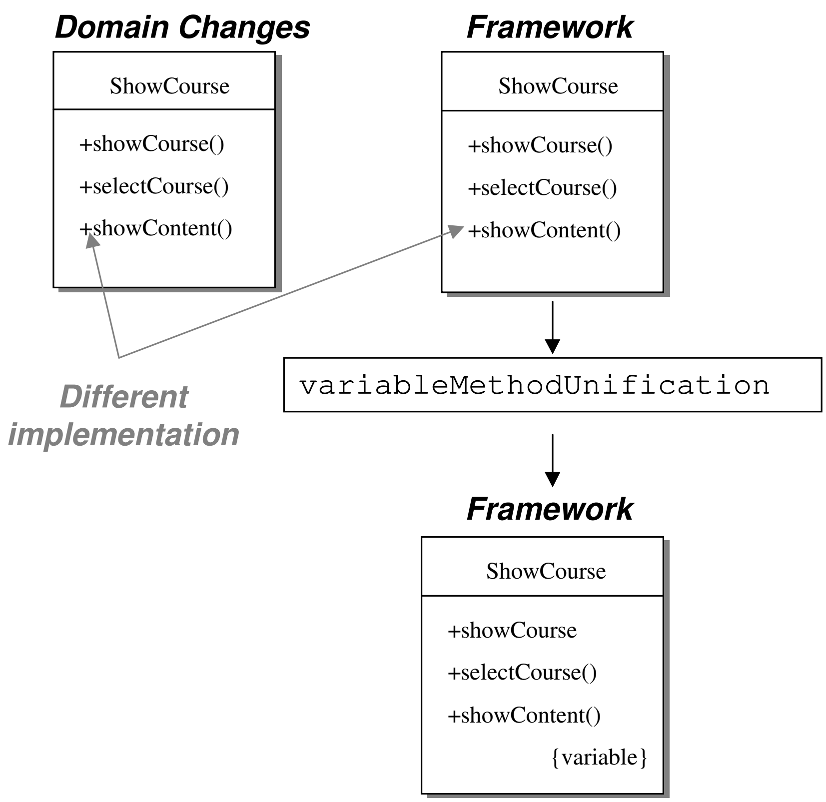

Figure 5. Unification rule example

Sometimes the application of refactoring procedures before the unification can improve its result. In this same example, it might be the case that the *showContent()* method varies only when displaying the course content but is stable when getting the content. Then, if the splitting refactoring had been applied before the unification the variation hot-spot would be *displayContent()* only, and not the entire *showContent()* method.

Framework instantiation normally takes place by completing the hot-spots in the framework. 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 [2]. 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:

- Variable method unification: is applied when an instance must change the implementation of a kernel (or frozen) method. After this transformation is applied the method is defined to be a variable method hot-spot. This unification has already been illustrated in Figure 5;
- Extensible class unification: is applied when an instance must add new methods to a kernel class. After this transformation is applied the class is defined to be an extensible class hot-spot. Figure 6 illustrates the use of this rule;
- Extensible interface unification: is applied when new class must be added to a given framework instance, and this addition is not supported by any of the framework extensible interfaces. After this transformation a new extensible interface is created. It can be a new class/interface or it can be one of the classes/interfaces already defined in the framework structure. One example of the use of this rule is presented in Figure 7.

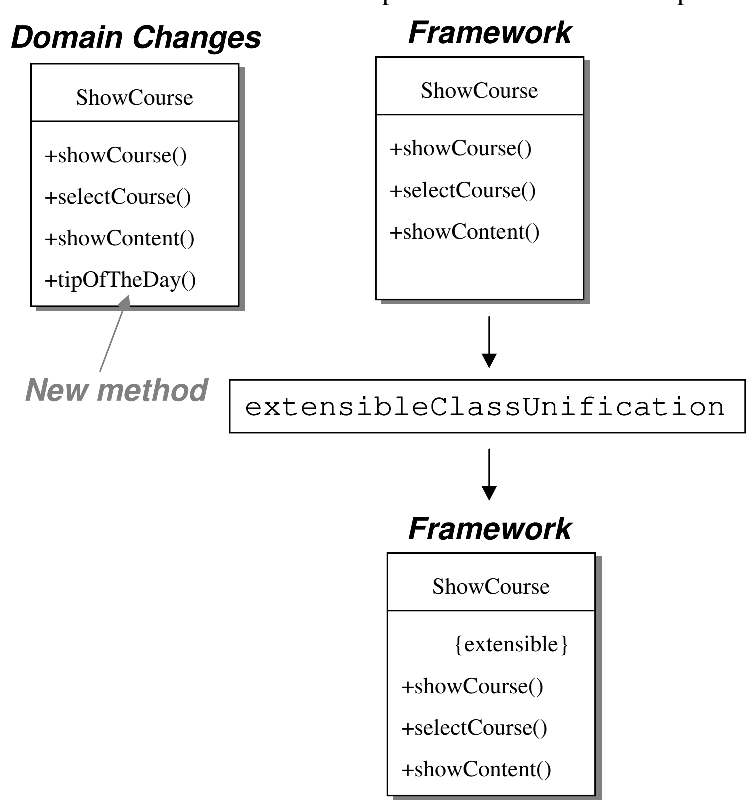

Figure 6. Extensible class unification example

In [6] we describe a tool that supports the definition of UML-FW projects and the application of refactoring and unification rules. The application of unification transformations has to be assisted by a human actor since several parameters need to be provided. The tool uses the formal semantic model proposed in [6] to verify the transformations.

Unlike refactorings, unification rules may only be supported if the framework is represented through a design language that explicitly represents its hot-spots.

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 8 illustrates this process.

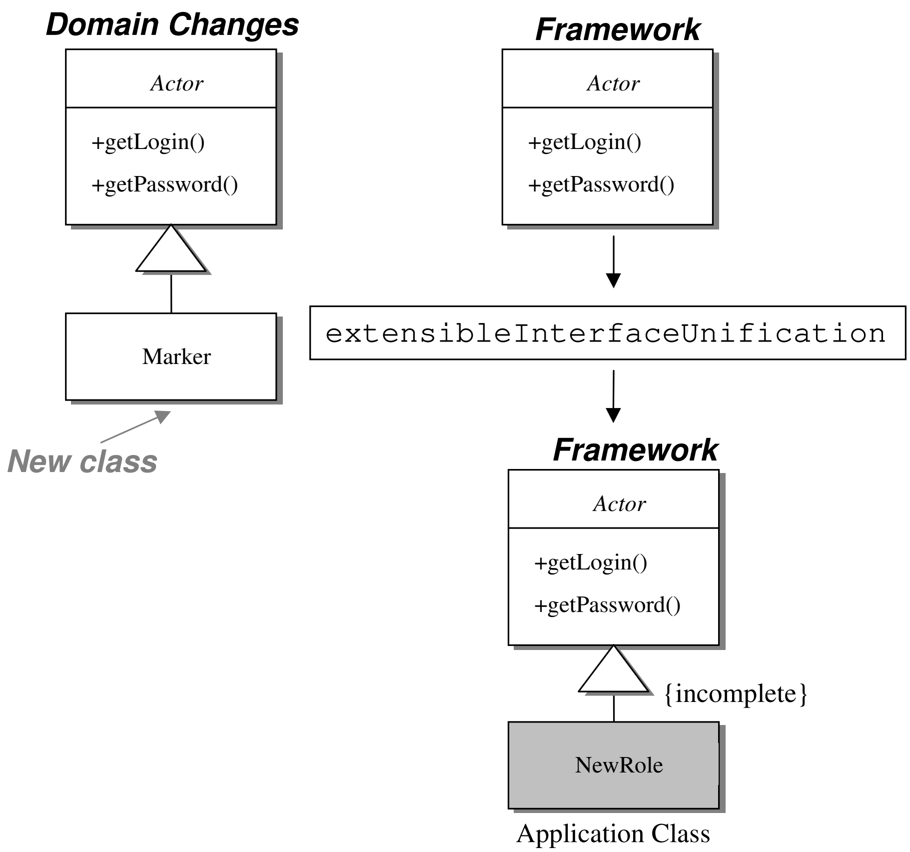

Figure 7. Extension interface unification example

The ALADIN framework [8] was defined using this approach, as described in [7]. The unification transformations illustrated in section 3 have been applied as part of the ALADIN development process. 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.

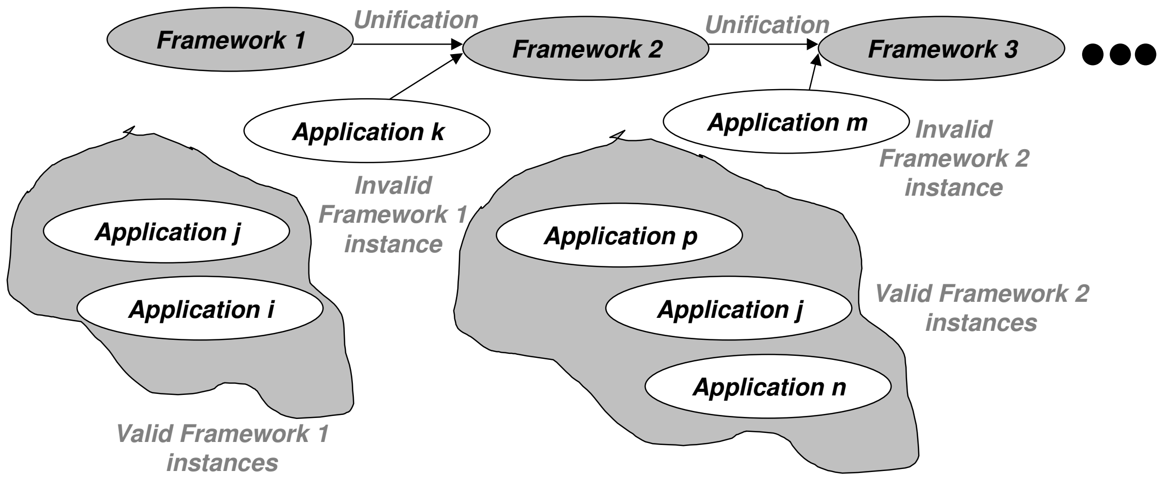

Figure 8. Unification-based software development

5. RELATED WORK

Currently there are very few framework design methods. A pattern-based description of some accepted approaches underlying framework design can be found in [20]. However, existing methods do not address the problem of identifying the possible relationships between kernel and hot-spot elements. Other interesting aspects regarding framework design such as framework integration, version control and over-featuring can be found in [2].

The Refactoring Browser [19] 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 [20]. They claim that framework abstractions can be derived from concrete applications. The unification-based development process may be used to systematize this approach.

A model for framework development based on viewpoints is proposed in [7]. 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. The UML-FW design notation helps the application of these transformations since it explicitly represents the hot-posts in the framework design. Several cases studies of the application of UML-FW to assist framework design and instantiation can be found in [6, 9].

A supporting tool is described in [6]. A new version of this tool that provides cooperative work capabilities and graphical representation is now being developed in Java. This tool will be used to experiment with different objet-oriented models and evaluate the impact of these models to framework design. More concretely, the tool will provide an uniform way of validating how roles [16, 17], and ADVs [3] can enhance framework design since unification and refactoring rules that can profit from these models will be compared through the environment.

Unifications are transformations used to avoid the architectural drift problem [2] by restructuring the framework hot-spots. After the application of a unification transformation the set of applications that may be instantiated based on the framework is enlarged. 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 [20].

REFERENCES

1. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

2. 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.

3. D. Cowan and C. Lucena, "Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse", *IEEE Transactions on Software Engineering*, 21(3), 229-243, 1995.

4. D. D'Souza, A. Sane, and A. Birchenough, "First-class Extensibility for UML – Packaging of Profiles, Stereotypes, Patterns", UML'99, *LNCS 1723*, Springer-Verlag, 265-277, 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, "A Systematic Approach for Framework Development", Ph.D. Thesis, Computer Science Department, PUC-Rio, 1999 (http://www.cs.princeton.edu/~fontoura).

7. 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*, 2001 (accepted for publication, to appear).

8. 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 (also submitted to WWW Journal).

9. M. Fontoura, W. Pree, and B. Rumpe, "UML-FW – A Modeling Language for Object-Oriented Frameworks", Technical Report 613-99, Princeton University, 1999 (also submitted to ECOOP 2000).

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

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

12. OMG, "OMG Unified Modeling Language Specification V.1.2", 1998 (http://www.rational.com/uml).

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

14. D. Parnas, P. Clements, and D. Weiss, "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering,* SE-11, 259-266, 1985.

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

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

17. D. Riehle and T. Gross, "Role Model Based Framework Design and Integration", OOPSLA'98, *ACM Press*, 117-133, 1998.

18. A. Riel, *Object-Oriented Design Heuristics,* Addison-Wesley, 1995.

19. 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/).

20. 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.

21. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Clifs, 1994.

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

23. J. Vlissides, *Pattern Hatching: Design Patterns Applied*, Software Patterns Series, Addison-Wesley, 1998.