JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 8. November-December 2006

Framework Evolution Tool

Mariela Cortés, Computer Science Department. State University of Ceará, Av. Paranjana 1700, Itaperi – 60.740-020, Fortaleza, CE, Brazil

Marcus Fontoura, Computer Science Department. IBM Almaden Research Center, 650 Harry Road, 8CC/B1, San Jose, CA, 95120, USA.

Carlos Lucena, Computer Science Department. PUC-Rio, Rua Marquês de São Vicente, 225 – 22453-900, Rio de Janeiro, RJ, Brazil.

Abstract

Framework development is very expensive, not only because of the intrinsic difficulty related to capturing the domain knowledge, but also because of the lack of appropriate methods and techniques to support the evolution of the framework architecture. Extension rules in combination with the refactoring approach can be used to support framework evolution, assuring consistency with the previously instantiated applications. In this paper we present a refactoring tool that has been extended to support most complex refactorings to implement the extension rules. This feature allows framework redesign and evolution that are consistent with its application instances.

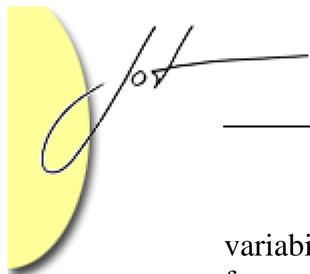
1. INTRODUCTION

Over the past years, framework technology has become a common reuse technology in object oriented software development. The major reason is to introduce a software development approach where entire software families are developed as opposed to stand-alone software applications.

A framework is an extensible semi-finished piece of software that represents a generic solution to a set of applications in a specific domain. A framework is composed of a *kernel* subsystem, which is common to all the applications that may be generated within the framework, and *variation points,* which represent areas of variability within a framework that can be adapted or extended to provide application specific behavior **[Codenie97].**

An intrinsic property of software in a real-world environment is its need to evolve. In particular, framework technology constitutes an ever-evolving representation of our knowledge of the domain in terms of variations and commonalties. A key point in framework development is that design work should not start by trying to model its

Cite this article as follows: Mariela Cortés, Marcus Fontoura, Carlos Lucena: “Framework Evolution Tool”, in *Journal of Object Technology*, vol. 5, no. 8, November-December 2006, pp. 101 - 124 http://www.jot.fm/issues/issue_2006_11/articlex



variability and flexibility at once. Instead, a fixed application should be designed from the framework domain; and it should be generalized only when the fixed case is understood **[Schmid99]**, as it is used. Clearly, such evolution affects existing instantiations, which need to be updated in a consistent way. Because of this, high costs are devoted to the maintenance of products built with the framework. In the context of framework evolution, refactoring and extension techniques are used to improve the quality of the framework. In **[Cortes05],** specialized extension rules are proposed to support the framework evolution consistent with its instance applications.

Although it is possible to refactor manually, this activity can be tedious and error-prone. Consequently, tool support is considered crucial. A tool provides information management that makes it possible to make better and well-informed decisions about the framework's evolution, especially with respect to the following issues: identification of evolution-prone modules and change impact analysis. Extension rules can be visualized as a composition of primitive refactorings. Since each primitive refactoring preserves the program behavior, the entire composition is itself behavior-preserving. Today, a wide range of tools are available that automate various aspects of refactoring[1]**.** Much of the prior work has focused on developing a small, primitive set of refactorings. However, the extension activity to support framework evolution is not supported by available tools.

The remainder of this paper is organized as follows: Section 2 describes the evolution processes. Section 3 presents the JRefactory tool. Section 4 details the incorporation of new features into the JRefactory tool to support extension rules. Section 5 describes a utilization sample of the tool. In Section 6 we comment on some related works. Finally, Section 7 concludes and presents future research directions.

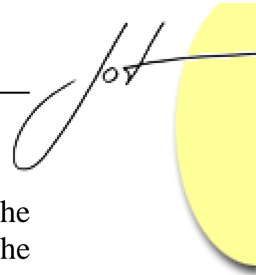
2. EVOLUTION PROCESS

In **[Butler01]**, framework development is considered equivalent to evolution process involving the execution of two tasks: restructure and extension. *Refactoring* is the object-oriented variant of restructuring: *"the process of changing a [object-oriented] software system in such a way that it not alter the external behavior of the code, yet improves its internal structure"* **[Fowler99]**. Complicated changes to a program can require both refactorings and extensions. In **[Cortés05],** the refactoring technique **[Opdyke92, Fowler99]** and extension rules are used**,** respectively, to support the execution of these tasks. We consider both techniques as *evolution processes* that are used to restructure the code and to add new abstractions preserving the observable behavior of the original design[2]. These techniques are very useful in developing efficient and flexible application frameworks and they fit well into the iterative framework development process.

Evolution processes may be used to avoid the architectural drift problem **[Codenie97]** by changing the variation point structure of the system. This phenomenon

[1] For an extensive and up-to-date overview of refactoring tools, we refer to http://www.refactoring.com/

[2] In computational sense, this implies that these transformations always result in legal programs equivalent to the original program.



occurs when the framework does not support the required customization and the application developers need to violate its structure. When the design modeled by the application is not a valid instance of a framework, evolution processes may be applied to add flexibility into the framework design, therefore, promoting its reutilization. The framework flexibility is based on the variation point structure that can be accessed by the users (application developers). Evolution processes can be considered behavior-extending transformations, since the cardinality of the application set can be increased after its application. The implementation of refactoring and extension rules to evolve framework designs into an evolution tool is the key point of this work.

2.1. Refactorings

Refactoring of source code **[Opdyke92, Fowler99]** is a well-known approach suggested for the development and evolution of frameworks by restructuring a program in a way that allows other changes to be made more easily. In specific situations, refactorings can be useful to implement extensions into the framework design, creating variation points **[Fayad99]**. The mechanics consist of turning the common behavior (kernel) into an alternative one that is encapsulated into prefabricated classes. However, refactorings are not sufficient to deal satisfactorily with all possible extension processes.

The refactoring activity involves redesign of a program unit to take advantage of good practices in design, such as design patterns, to improve it. A natural relation between patterns and refactorings is presented in the design patterns catalogue **[Gamma95]**: *"Patterns... supplies targets for your refactorings"*. In other words, refactorings let designers focus on patterns when they are developing software projects. Patterns can be added through refactorings: *"... refactorings turn explicit the design patterns that are subjacent into the code"*. The use of design patterns has costs related to complexity and indirection. For this reason, design should be as flexible as needed, not as flexible as possible. Refactorings have been shown to directly implement certain design patterns **[Tokuda01].** Examples of refactorings with this property are Replace *Type Code with State/Strategy* and *Form Template Method* **[Fowler99]**.

In counterpart, there are no refactorings for all design patterns (for example, no refactoring addresses the incorporation of design patterns using recursive subsystems). This incompleteness in the refactoring catalogue **[Fowler99]** can be solved through the use of extension rules, which are based on metapatterns. Metapatterns model all the possible combinations of template and hooks methods, including the recursive composition. These combinations are used to model framework variation points.

2.2. Extension Rules

Extension rules are used to extend the framework behavior, making it possible to instantiate a greater number of applications. These rules implement transformations that alter the framework variation point structure.

The variation point structure introduces variability that is transparent outside the subsystem, either by inheritance or by composition. Using extension rules during the

evolution process, two situations are possible: whether the base class of the variation point subsystem is introduced as a new class, or whether the responsibility of an existing class is extended by the responsibility of the base class. In the first case, we speak of an expanding transformation since we have expanded the original class structure by a new class. In the second case, we speak of an extending transformation, since we have extended an original class by new responsibilities.

Extension rules are based on framework metapatterns **[Pree94]**, which implement variation points as a combination of template and hook methods **[Gamma95, Pree96]**. A *template method*[3] provides the skeleton of behavior. A *hook method* is called by the template method and can be tailored to provide different behaviors. Currently, there are four extension rules that automate the incorporation of the basic patterns proposed by Pree **[Pree94]**: *Add Hook Method*, *Add Unification Pattern*, *Add Separation Pattern* and *Add Recursive Pattern*. In Section 4, we present the extension rule by means of short descriptions and the solution implemented into the JRefactory tool.

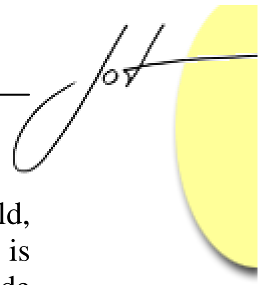
3. JREFACTORY TOOL

JRefactory[4] is an open source tool provided under the GPL written in Java to allow easy application of semi-automatic refactorings. The tool supports the application of the following refactorings:

- Repackage or move class
- Rename class
- Add an abstract parent class
- Add a child class
- Removes a class
- Push up field
- Push down field
- Push up method
- Push up abstract method
- Push down method
- Move method

JRefactory can be integrated directly into industrial strength IDEs and it also can be used in command line variant. The tool carries through the formatting of the code source (Java) received as entered. On the basis of the supplied Java language grammar, a parse tree representing the code structure is generated. This information is used to generate summaries and metadata from the code. On the basis of this excellent information the graphic user interface (GUI) is created on the basis of UML class diagram **[Rumbaugh98]**.

[3] Template method must not be confused with the C++ template construct, which has a completely different meaning.
[4] http://jrefactory.sourceforge.net



Refactorings in design level are applied by selecting the target element (field, method, and class) in the UML diagram. Since the appropriate refactoring in the menu is selected, the transformation is automatically applied. Differently, refactorings in code level are applied by the selection of the appropriated code from the IDE screen and the selection of the correspondent refactoring through the standard interface.

Both possibilities for interfacing make it possible to deal with refactorings in both design and code levels. In counterpart, the utilization of two different interfaces to apply the refactoring transformations introduces the need to keep both artifacts (class diagram and source code) in sync. Furthermore, functionalities of JRefactory tool provide the ability to format and print both UML class diagrams and Java code. UML class diagrams can be resized and are useful for navigating through lots of code. In particular, the GUI allows zooming (in certain steps), moving classes and changing association lines. The major modules of JRefactory are showed in the package diagram (Figure 1), indicating `<<import>>` dependencies.

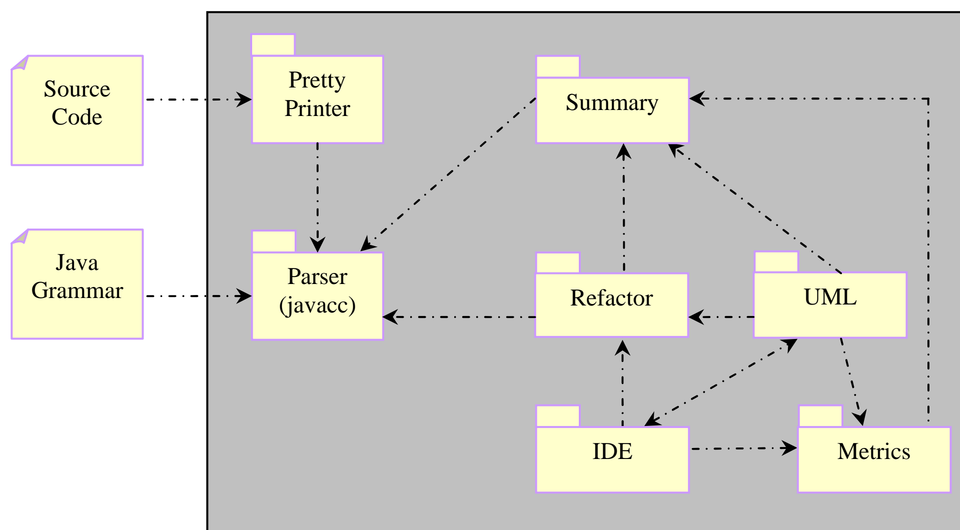


Figure 1. The major modules of JRefactory tool.

Another interesting feature supplied by the tool is support for metrics. It is possible to gather metrics about the java source code about package or class, including absolute and average numbers of classes, methods, and statements, each averaged for the higher structural units (e.g. average statements per class, or per method). The metric information highlights problems in classes or methods where there are either too many methods or too many statements per method, for example.

Several characteristics affect the usability of a tool, such as automation, reliability, configurability, coverage, and scalability **[Mens04]**. The degree of automation of a refactoring tool varies, depending on the refactoring activities that are supported. The JRefactory tool includes a refactoring browser that supports a semiautomatic approach to refactoring. While it remains the task of the developer to identify which part of the

software needs to be restructured and to select the most appropriate refactoring to apply, the actual application of the refactoring is automated.

The reliability of a refactoring tool mainly depends on the ability to guarantee that the provided refactoring transformations are truly behavior preserving. In this sense, JRefactory checks specific refactoring preconditions before applying it. In addition, an *undo* mechanism is provided.

In the context of configurability and openness, JRefactory can be easily integrated into industrial strength IDEs. This is typically achieved using the built-in extensibility mechanisms of these tools (e.g., plug-ins, APIs, or wizards). Moreover, JRefactory is an open tool that allows the addition of new refactorings and definition of composite refactorings from primitives ones. This refactoring composition increases the scalability and performance of the tool.

4. JREFACTORY EXTENSIONS

The JRefactory tool was enriched with the following new features to support the extension rules presented in Section 2.2: *Add Hook Method, Add Unification Pattern, Add Separation Pattern,* and *Add Recursive Pattern*. In Figure 2, an updated diagram is presented in which the tag stereotype `<<application>>` marks packages that had been modified or incorporated into the JRefactory tool.

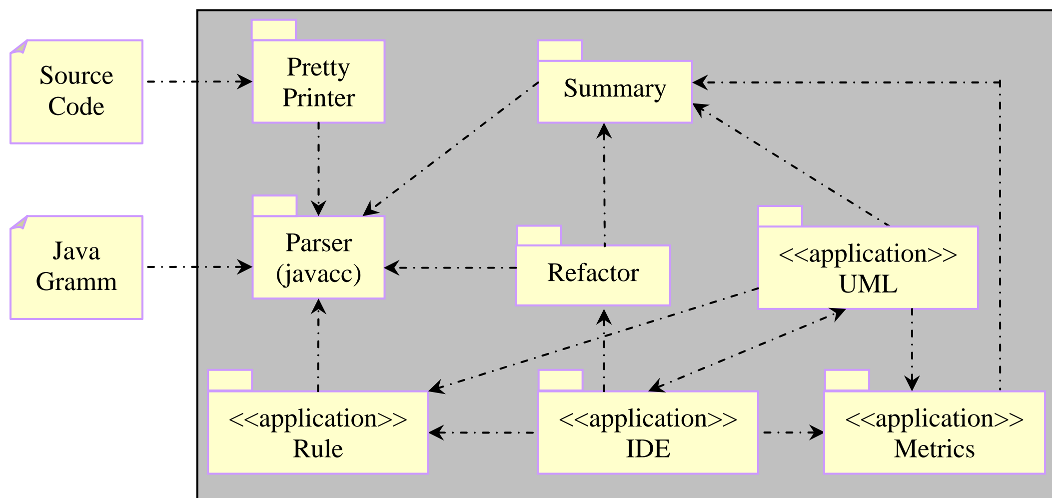
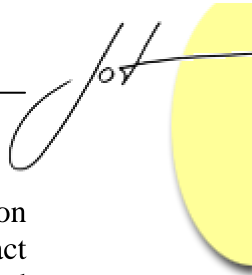


Figure 2. Upgrade of JRefactory to support extension rules

The upgrade of the JRefactory tool was carried out in three stages:

1. Modification to IDE interface, mostly from the graphical interface (GUI) to allow access to the rules.
2. Implementation of the extension rules.



3. Incorporation of additional metrics that help verify the impact of the evolution processes on the framework design, as, for example, the number of abstract methods — since framework variation points generally are implemented through them.

The first stage involves the manipulation of menus in the JRefactory GUI. Using this interface, refactorings are associated with the active component (class, method or field) in the class diagram. Refactorings applied to classes are always accessible; however, refactorings for methods and fields are accessible only if the respective element is active in the diagram. The application of extension rules from the GUI is restricted to the application of *Add Hook Method* and *Add Recursive Pattern* rules, since only the selection of the respective method is needed. This selection can be realized from the class diagram. Other rules require code selection that only can be performed from the IDE to complete their execution. All extension rules must be raised from the IDE menu

Refactorings are implemented in classes grouped into packages. Classes implementing refactorings are defined as specialization of the abstract class *Refactoring* into the *Refactor* package. The main method in the *Refactoring* class that carries out the transformation is *run()* (Figure 3).

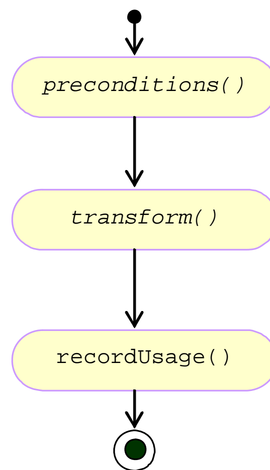


Figure 3. Activity diagram of run() method.

The execution of the *run()* method involves checking of pre-conditions. If all pre-conditions are satisfied, the transformation is realized. Otherwise, the execution is aborted and an exception is raised. Finally, a log file is generated with the information about the refactoring execution. The *run()* method is a template method, which represents the execution skeleton for all refactorings in three steps: *preconditions(), transform()* and *recordUsage()*. The hook methods *preconditions()* and *transform()* are defined in subclasses to realize the specific behavior. Extension rules are implemented following the same structure of standard refactorings: as specialization of the base class *Refactoring*. In this way, the implementation of an extension rule is as shown in Figure 3.

Framework design implies the existence of a set of instance applications that are generated from it. In the context of framework evolution, the process introduces extra complexity, since application instances must be considered to ensure consistency with the framework, thus avoiding the architectural drift **[Codenie97]**. Extension rules are defined to support iterative development and framework evolution. In this way, the evolution of instance applications must be considered during the application of the extension rules. In the next sections, a detailed description of the implementation of each rule into the tool is presented. In the following section, we present each extension rule through a short description and the solution proposed and implemented into the JRefactory tool. Illustrations are presented in terms of UML class diagrams. In addition, we use visual representations for the UML-F tags for *framework, application and patterns* **[Fontoura01]**.

Add Hook Method Rule

Description. This rule is used to incorporate a hook method into the framework design to implement a new variation point.

Solution. The instance application needs to change the implementation of a kernel method. In this way, each application can define alternative behaviors. Figure 4 illustrates this process.

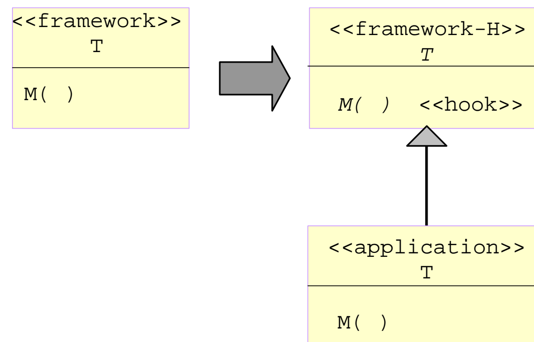
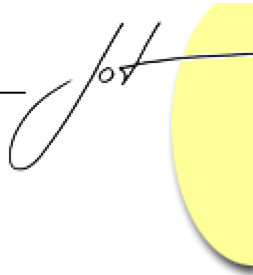


Figure 4. Application of Add Hook rule

Implementation. AddHookMethodRule class implements the rule, which is a tail of an inheritance tree rooted in *Refactoring* class. Specifically, the rule extends the abstract class *AddClassRefactoring* (Figure 5).



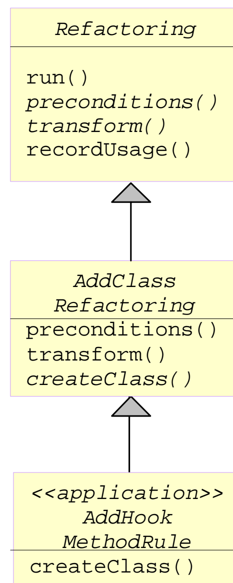


Figure 5. Class diagram for the inheritance tree of AddHookMethodRule.

Typically, several concrete classes must be created before the correct abstraction becomes apparent. *AddClassRefactoring* class partially implements a refactoring to incorporate a new class into the system. This class is used as the base for other classes to implement a specific behavior in order to add a new class under particular conditions. For example, the already existent *AddChildRefactoring* subclass implements a refactoring to add a new class as the child of a specific one.

AddClassRefactoring extends the base class *Refactoring*; thus, the *run()* method displays the structure showed in Figure 3. *Preconditions()* method checks the valid application for the refactoring, verifying if the new class already exists. The template method *transform()* deferring the creation of the new class to subclasses. Consequently, subclasses must define the hook method *createClass()* to implement this step.

The hook method *createClass()* in the *AddHookMethodRule* subclass is depicted in the following activity diagram (Figure 6). The activities involve the insertion of an empty class as a subclass of an existing and selected one (candidate for hook class in the framework). In sequence, the system's metadata is updated.

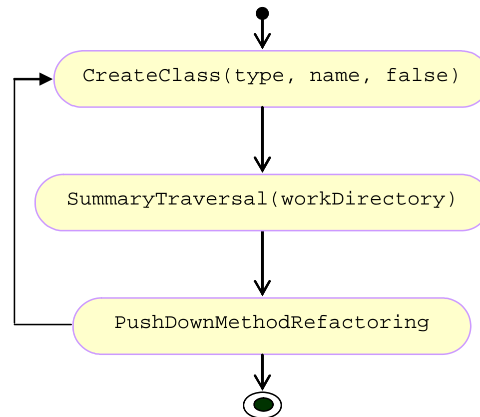


Figure 6. Activity diagram for the createClass() method in Add Hook Method rule.

Next, the target method for the *Add Hook Method* rule application, selected from the super class, is pushed down to the recently created subclass, remaining abstract in the base class. To realize this step, the *Push Down Method* refactoring, already existent in the tool, is used. The repetition of this sequence of activities occurs for each instance application. In this way, the consistency of the instance applications is preserved, since the original behavior of the framework core is transferred to classes of instance applications. The abstract method in the superclass represents a new framework variation point.

Add Unification Pattern Rule

Description. This rule is used to incorporate the *Unification* pattern into the design. This pattern occurs when both the template and hook methods belong to the same class.

Solution. Variant steps are implemented as combinations of template-hook methods. The hook method executes the special behavior required by the application developer. The method might be created through the *Extract Method* refactoring **[Fowler99]**, which replaces a fragment of code with a call to the newly created method (Figure 7).

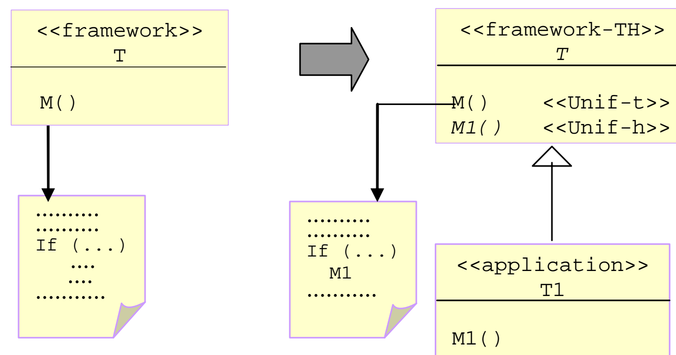
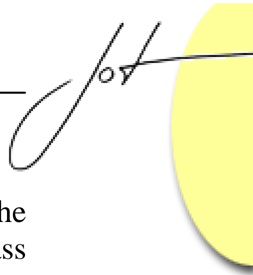


Figure 7. Application of *Add Unification Pattern* rule



Implementation. *Add Unification Pattern* rule is implemented as a specialization of the abstract class *Refactoring* (Figure 8). In this way, the abstract methods in the superclass *preconditions()* and *transform()* must be implemented.

Preconditions for the application of this rule consist of the verification of properties about the integrity of the selected fragment of code.

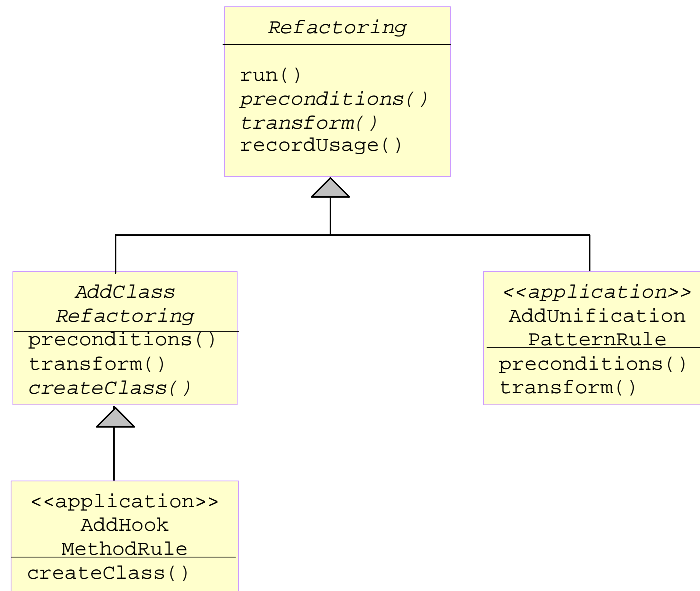


Figure 8. Class diagram for the inheritance tree of Add Unification Pattern rule.

After the precondition, checking the transformation process is initiated. The execution of the *transform()* method involves the activities showed in Figure 9.

The creation of the new method from a portion of an existent method in the framework core is carried out using the *Extract Method* refactoring. After the extraction and since the metadata have been actualized, the new method must be transformed into a hook method. In this sense, the evolution process is similar to the *Add Hook Method* rule involving the creation of a class hierarchy to represent the specific application classes, and moving down the method through the hierarchy.

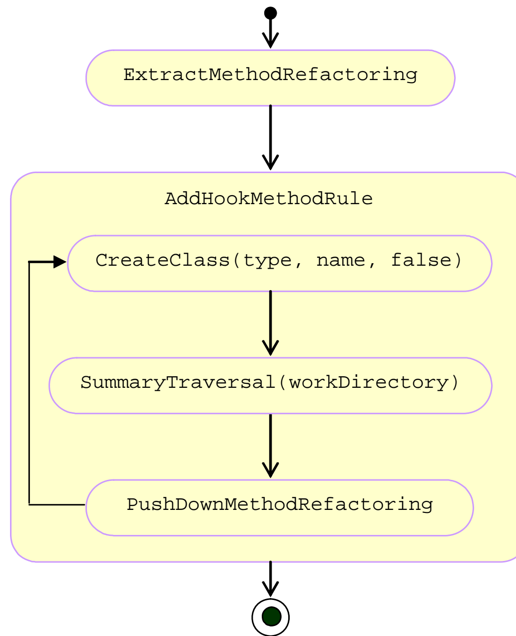


Figure 9. Activity diagram for *transform()* method for *Add Unification Pattern* rule

As a result, the concrete class in the original framework design is transformed into a template-hook class; meanwhile, the concrete subclasses implement the original behavior for each instance application.

Add Separation Pattern Rule

Description. This rule is used to incorporate the *Separation* pattern into the design. This pattern occurs when template and hook methods belong to different classes. The application of this rule transforms a fragment of code from a method in the core into a hook method.

Solution. Create a new variation point method in a separate hook class (Figure 10). This variation point must be extended by composition. In the obtained design, an additional class is required to host the template method upon adding to the variation point subsystem.

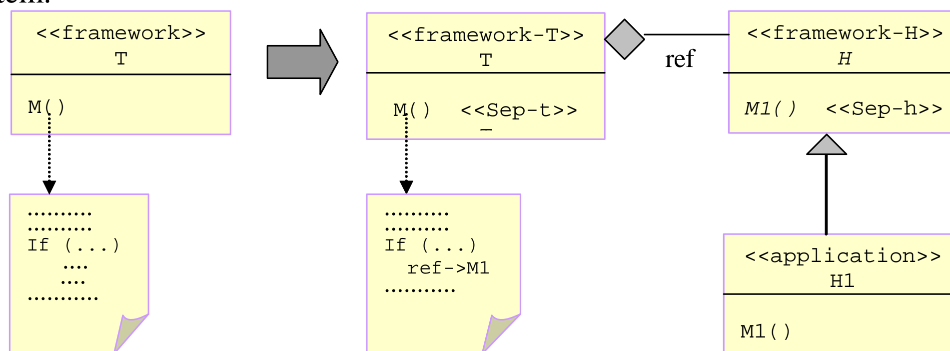
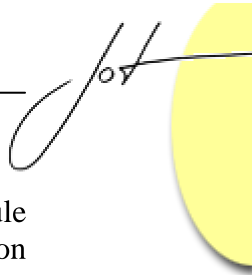


Figure 10. Add Separation Pattern rule



Implementation. Similar to *Add Unification Pattern,* the *Add Separation Pattern* rule transforms a fragment of code from a method from the framework core into a variation point. The main difference is that the variation point is created as a member of a separated class. Consequently, the extension rule transforms the concrete class in the original design into a template class and a new hook class is added into the framework core.

The *Add Separation Pattern* rule is defined as a specialization of the base class *Refactoring* (Figure 11). Since this rule is a composition of basic refactorings already existing in the JRefactory tool, preconditions for its execution are partially checked for each refactoring that comprises the rule.

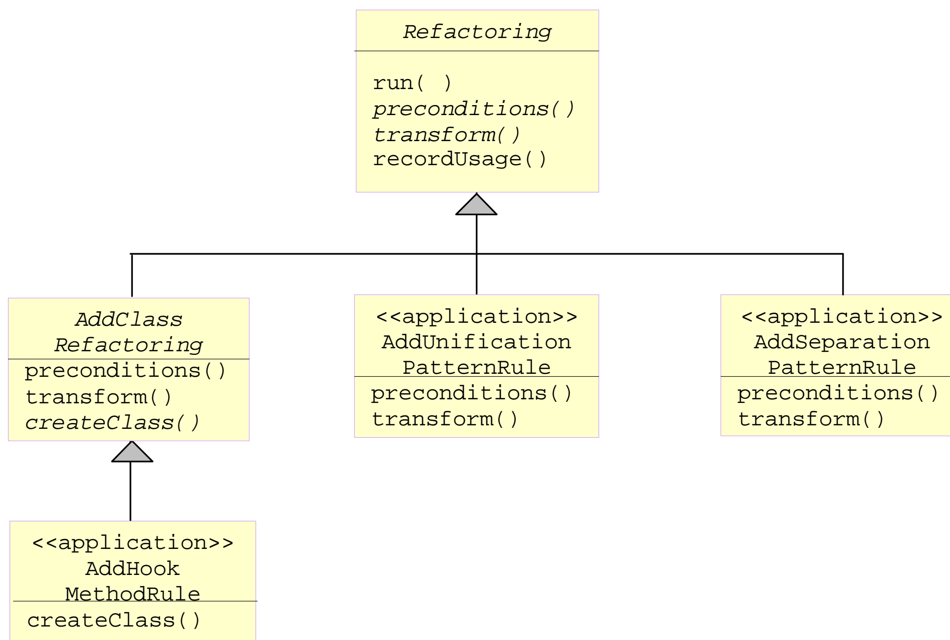


Figure 11. Class diagram for the inheritance tree of *Add Separation Pattern* rule.

The *transform()* method (Figure 12) that implements the rule involves the execution of *Extract Method* refactoring. This refactoring is responsible for extracting the selected fragment of code and transforming it into a new method, as a member of the original concrete class in the framework core. After that, a new separate class is created and the extracted method in the first step is moved into the new class using *Move Method* refactoring, already existent in the tool. In the current situation, *Add Hook Method* rule can be applied to transform the recently created class into a hook one. All changes are reflected in metadata and summaries of code using the *SummaryTraversal* method.

Finally, the relationship between the template and hook classes is established using *Add Field* refactoring, already existent in the tool. The refactoring is applied to the template class to create a reference for an object in the hook class. Both template and hook classes belong to the framework core.

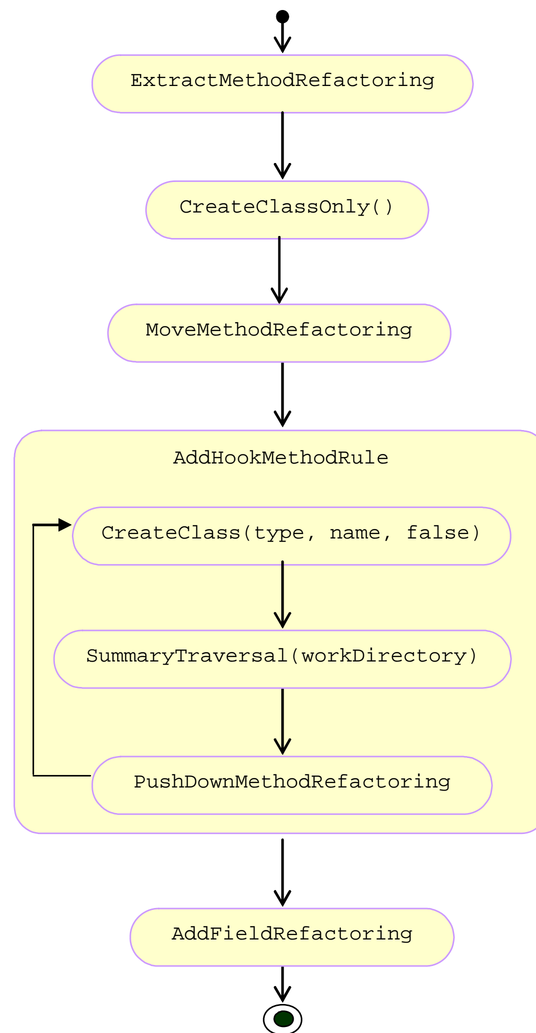


Figure 12. Activity diagram for *transform()* method for *Add Separation Pattern* rule

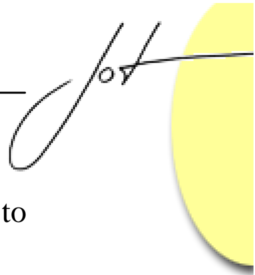
Add Recursive Pattern Rule

Description. This rule is used to incorporate the *Recursive* pattern into the design. This pattern occurs when an object in the template class refers to objects in the hook class. In particular, the template class is a descendent of the hook class.

As a result, the *Add Recursive Pattern* rule creates a *composite* class to implement a recursive variation point. The composite class extends a base class already existing in the framework core that represents an interface for objects.

Solution. Create an object composition to handle object collections in order to selectively add or modify behavior to instances (Figure 13).

In the *Recursive* pattern design any number of template classes can be defined as subclasses of H. These template classes can define additional/modified behavior. Note



that any number and combination of instances of template classes can be attached to instances of H descendants.

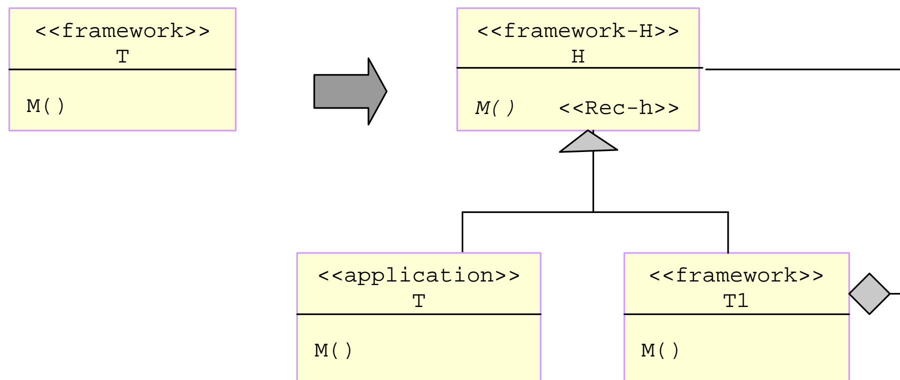


Figure 13. Application of *Add Recursive Pattern* rule

The obtained design is coincident with the structural pattern *Composite* [Gamma95]. Thus, the application of this rule is useful for incorporating the corresponding pattern into the design.

Implementation. Similar to the *Add Hook Method* rule, the *Add Recursive Pattern* rule is implemented as a specialization of the abstract class *AddClassRefactoring* (Figure 14). The template method *transform()* in the super class, lets its subclass redefine the hook method *createClass()* to implement the specific behavior.

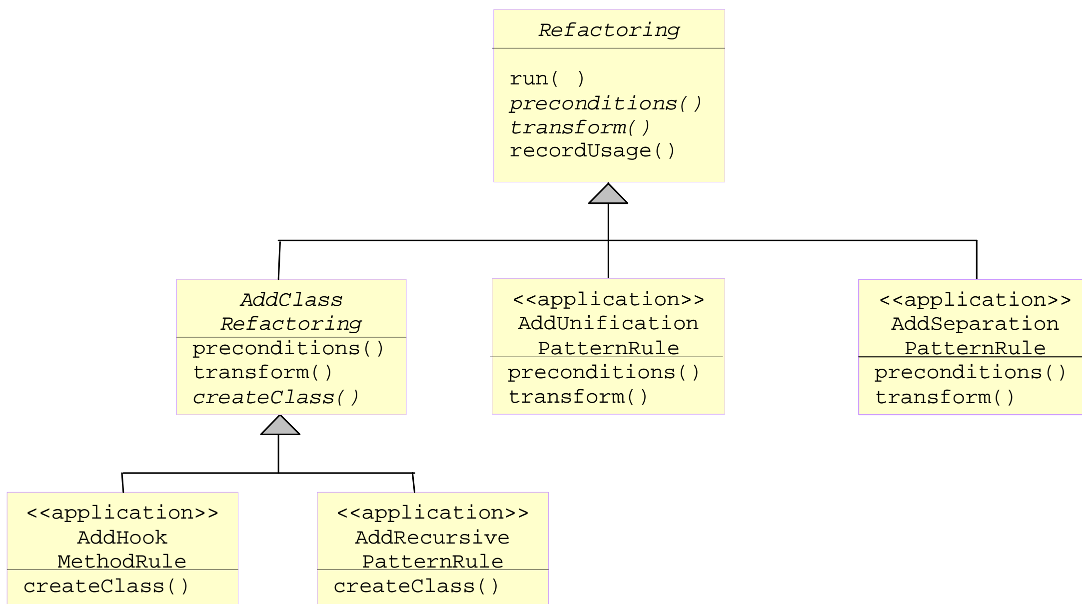


Figure 14. Class diagram for the inheritance tree for *Add Recursive Pattern* rule.

The hook method *createClass()* is implemented in the *AddRecursivePatternRule* subclass as depicted in the following activity diagram (Figure 15), which involves the creation of a new class for each instance application. Each class is created as a subclass of an existing one into the framework core. In addition, an extra subclass is created representing a composite element, which is included in the framework core. The composite class maintains a collection of primitive components.

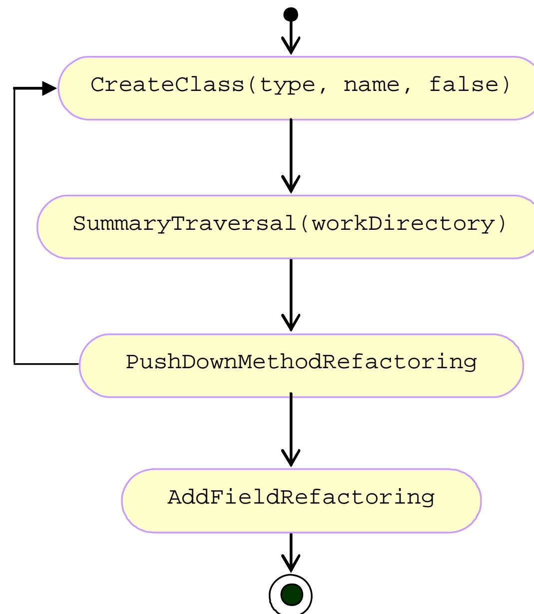


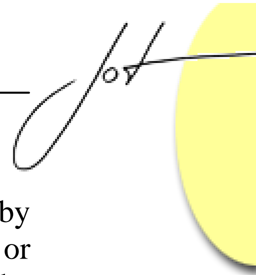
Figure 15. Activity diagram for *createClass()*method in *Add Recursive Pattern* rule

Since the inheritance tree is created and the system metadata are actualized, the selected method in the superclass is moved down to subclasses using *Push Down Method* refactoring, leaving it abstract in the super class. The abstract method represents a new variation point in the framework. In this way, the consistency with the application instances is preserved since the framework behavior is partially transferred to subclasses. Thus, the generic behavior in the framework core becomes a specific one in the instance applications, preserving the semantic consistency of the system.

Finally, a reference to an object from the superclass is added in the composite subclass to represent the object composition, using *Add Field* refactoring. This structural pattern allows the redirection of messages through the class hierarchy. Consequently, clients treat individual objects and compositions of objects uniformly.

Metrics

The JRefactory tool contains an interesting feature to extract metrics about the project code using updated summaries generated by the tool. JRefactory supports metrics about the complete project, package, class, and methods. Metrics can be calculated in absolute or relative values, on the basis of classes, methods and sentences. The results obtained are useful for establishing the code's properties, such as if the method number is too high or



if the methods are too long. Moreover, different versions of software can be evaluated by comparing the respective metrics. On the basis of this information, the user (developer or designer), can restructure the code using refactorings. The available version of the JRefactoring tool encompasses the following metrics:

Framework Structure Level: Metrics for assessment of framework structural characteristics.

- Design Size in Classes – DSC – count of the total number of classes in the system
- Number of Abstract Classes – NAC
- Number of Interfaces – NOI
- Number of Statements
-

Class Internal Level: Metrics assessing functional characteristics of individual classes.

- Number of Class Methods (NOM)
- Number of Public Methods (PubM)
- Number of Instance Variables (NIC)
- Number of Class Variables (NCV)

Method Level: Metrics assessing functional characteristics of individual classes.

- Number of Parameters per Method (NPM).

Metrics have a number of interesting characteristics for providing evolution support **[Mens01]**: *"They are simple, precise, general and scalable to large-size software systems."* Metrics can be used before the evolution has occurred (i.e., predictive), to analyze the software quality, and after the evolution has occurred (i.e., retrospective), to find out whether its structure of quality has improved. Alternatively, one can study the evolution process, e.g., to understand what has been changed and how.

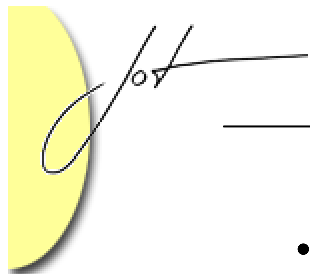
In this context, new metrics were incorporated into JRefactory to obtain a measure of the framework evolution process to assess whether the evolution goals have been achieved. These metrics support reasoning about the framework properties and evaluate the framework flexibility and adaptability. Thus, we implemented the following metrics into the JRefactory:

Framework Structure Level:

- Number of Single Inheritance (NSI) count of the number of classes (subclasses) that use inheritance in the system.
  Motivation. During the extension phase none of the classes changes their hierarchy nesting level, but many of them increase or decrease the number of children. Thus, all changes were made to the leaves of the inheritance hierarchy, which is indeed typical for extensions.

Class Internal Level:

- Number of Attributes (NIC + NCV)



- Number of Public Attributes of a Class (PubA)
  Motivation. Used to assess the encapsulation level, and thus allow reasoning of the flexibility and reusability levels.
- Number of Abstract Methods (NAM)
  Motivation. This metric can be used to assess functional characteristics of individual classes. A variation point is often implemented via a template method **[Gamma95, Pree96]**. A template method is implemented via a polymorphic method invocation of the hook method. Since object-oriented languages achieve polymorphism via method overriding, this suggests inferring variation points by analyzing overridden methods. Such methods in Java are marked as an *abstract* method.

On the basis of the information supplied from metrics, it is possible to establish design properties and consequences of the evolution processes. This feature offers immediate feedback after the refactoring application. For example, when an evolution rule is applied, the metrics about interfaces, methods and abstract classes must return a bigger occurrence index than found in the metrics before the application. This occurs because the extension rules incorporate new explicit variation points into the design, which are implemented using these elements.

5. JREFACTORY UTILIZATION

In this section we present some screen shots of the utilization of JRefactory tool in practice using the study case illustrated in **[Cortes05]**. The project imported into the JRefactory environment consists in a framework for web searching. The corresponding initial class diagram is shown in Figure 16.

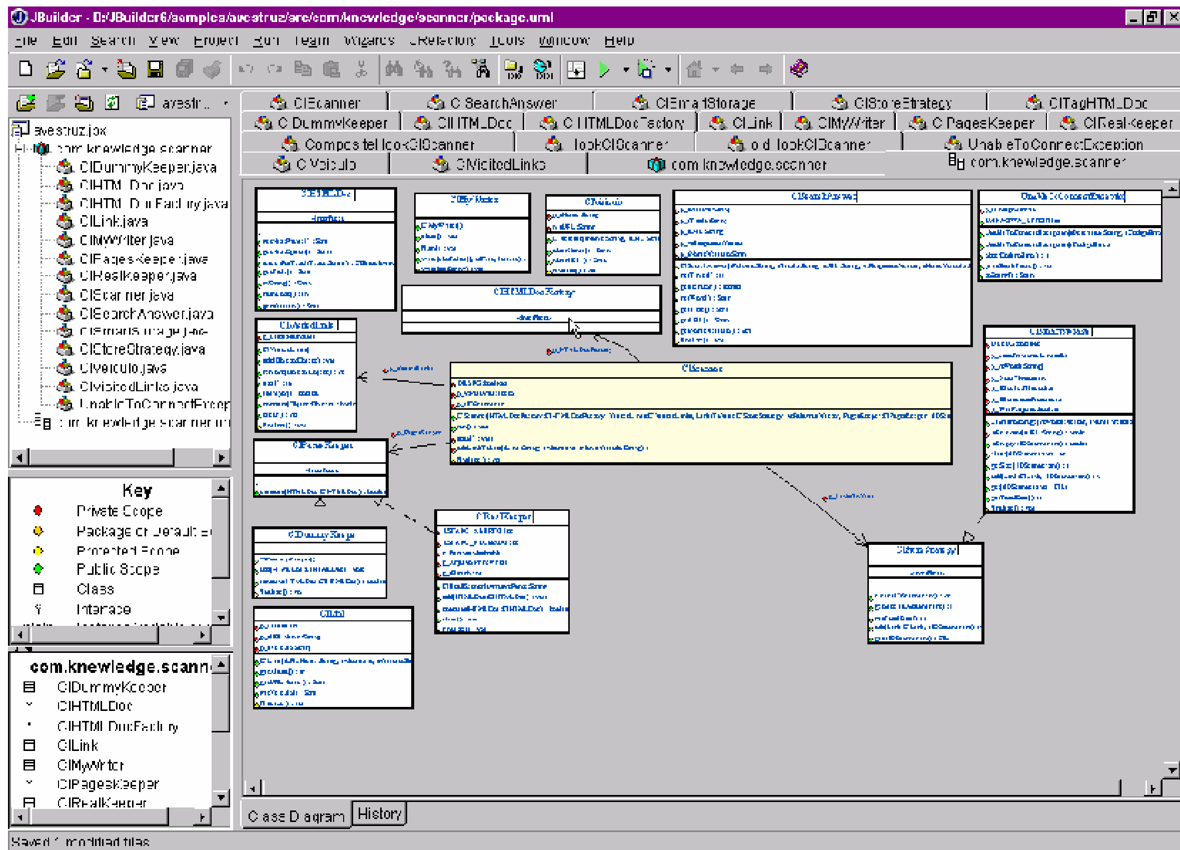


Figure 16. Screen shot of JRefactory GUI

Another screen shot from the JRefactory presents the browser content consisting of Java code from the JBuilder IDE. Using this interface, the application of *Add Unification* and *Add Separation Pattern* rules are feasible. Figure 17 shows the code selection for the application of *Add Separation Pattern* rule. Next, the tool opens a dialog box presenting the candidates parameters for the new method and requesting for information about method name, return values and protection properties. Finally, the signature for the method is presented.

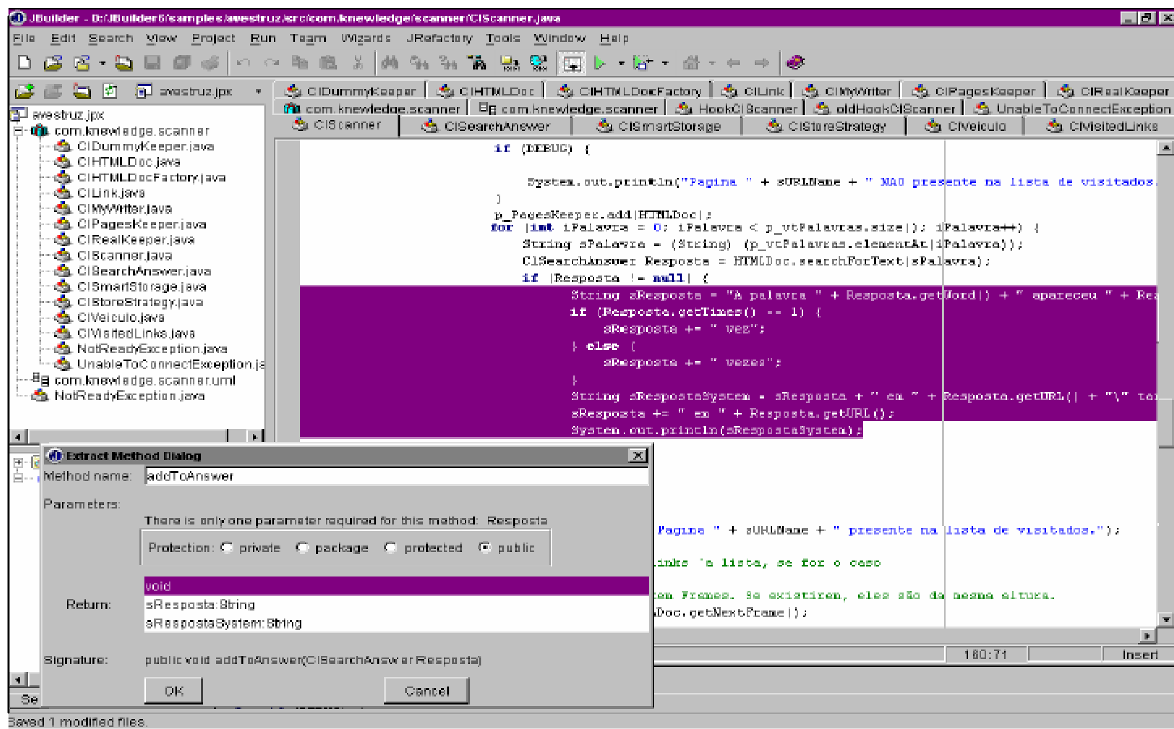


Figure 17. Screen shot of JRefactory browser during *Add Separation Pattern* rule application

The user confirmation raises the effective transformation that is reflected in the code. Consequently, the respective metadata and summaries are generated in a way that is consistent with this actualization. Figure 18 represents the final design after the application of this extension rule.

Figure 18. Screenshot of the design after Add Separation Pattern rule application

6. RELATED WORKS

Currently, there are very few framework design methods that deal with framework evolution. Refactoring is a useful technique that is easily carried out manually. However, tools can help refactoring proceed more quickly. The first refactoring tool built was the Smalltalk Refactoring Browser. The Refactoring Browser **[Roberts97]** is a tool to help maintain frameworks written in Smalltalk. It currently does not support extension rules, but it has an open architecture and the introduction of the rules and new refactorings seems to be straightforward. Metrics are not supported. Unfortunately, Smalltalk is not a mainstream commercial language.

Subsequently, several refactoring tools for other languages have emerged, in particular for Java. **[Florijin97, Meijers96, Winsen96, Gruijs97]** propose tools that provide extensive support for working with design patterns to achieve framework restructuring. The tools provide automated transformations that can be used to evolve design pattern instances in predefined ways at a high level of abstraction. These transformations are similar to our metapattern approach, but neither change propagation or application upgrading are addressed.

7. CONCLUSION

In this work we present an extension of an already existing tool for software refactoring to support evolution rules. The extension consists of the inclusion of four extension rules to support framework evolution: *Add Hook Method, Add Unification Pattern, Add Separation Pattern* and *Add Recursive Pattern*. Similar to refactorings, extension rules automate many common design transitions and reduce the likelihood of errors. While such transformations have been shown to support the introductions of metapatterns in object-oriented applications, we are aware of their usage to support the evolution of frameworks and guide developers when evolving the framework, by propagating the appropriate changes, as well as application upgrading.

Through the utilization of structural information in addition to formal methods to check the behavioral model it is possible verify the correctness of the evolution processes that are proposed. Using a combination of these techniques, it is possible to check if the processes preserves the external behavior, and establishes some properties regarding the flexibility of the evolving design.

There are other possible evolution rules that can be defined for variation points that are not implemented as template-hook combinations. In the future, we hope to elaborate and include new rules for different kinds of variation points in the tool.

Acknowledgments

This work is being supported in part by the FUNCAP and National Research Council of Brazil (CNPq).

REFERENCES

[Butler01] Butler G., Xu L. Cascaded Refactoring for Framework Evolution. *Proceedings of 2001 Symposium on Software Reusability*. ACM Press, p. 51-57. 2001.

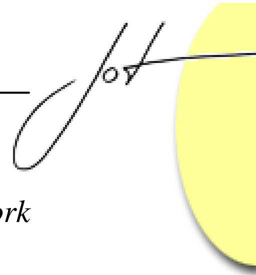
[Codenie97] Codenie W., Hondt K., Steyaert P., Vercammen A. From Custom Applications to Domain-Specific Frameworks. Communications of the ACM, 40(10): 71-77. 1997

[Cortés05] Cortés M., Fontoura M, Lucena C. Rule-Based Approach to Framework Evolution. To appear in: Journal of Object Technology. 2005.

[Chindamber94] Chindamber S., Kemerer C. A metrics suite for object-oriented design. IEEE Trans. Software Engineering, 20(6): 476-493, June 1994.

[Fayad99] Fayad M., Schmidt D., Johnson R. Application Frameworks. *Building Application Frameworks*. New York: Wiley. p. 3-28. 1999.

[Florijin97] Florijin G., Meijers M., Van Winsen P. Tool Support for Object-Oriented Patterns. LNCS 1241. Springer-Verlag, p. 472-495. 1997.



[Fontoura01] Fontoura M., Pree W., Rumpe B. *The UML Profile for Framework Architectures*, Addison-Wesley. 2001.

[Fowler99] Fowler M. *Refactoring: Improving the design of existing code*. Addison-Wesley. 1999.

[Gamma95] Gamma E., Helm R., Johnson R., Vlissides J. *Design patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

[Gruijs97] Gruijs D. A Framework of Concepts for Representing Object-Oriented Design and Design Patterns. Master's thesis, Utrecht University, 1997.

[Meijers96] Meijers M. Tool Support for Object-Oriented Design Patterns. Master's thesis, Utrecht University, 1996.

[Mens98] Mens T., Mens K. Assessing the architectural quality of evolve systems. In: Proceedings of ECOOP'98: Workshop on techniques, Tools and Formalisms for capturing and assessing architectural quality in object-oriented software. 1998.

[Mens01] Mens T., Demeyer S. Evolution Metrics. IWPSE 2001, Vienna, Austria. 2001.

[Mens04] Mens T., Tourwé T. A survey of software refactoring. IEEE transactions on software engineering, vol. 30, no. 2. 2004.

[Opdyke92] Opdyke W. Refactoring Object-Oriented Frameworks, Ph.D. Dissertation, Computer Science Department, University of Illinois, Urbana-Champaign. 1992.

[Pree94] Pree W. Mettapatterns- A Means for capturing the essentials of reusable object-oriented design. In: Proceedings, ECOOP'94. Springer-Verlag, Berlin, 1994.

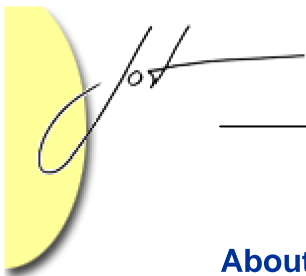
[Pree96] Pree W. Framework Patterns. New York City: SIGS Books. 1996.

[Roberts97] Roberts D., Brant J., Johnson R. A Refactoring Tool for Smalltalk. Theory and Practice of Object Systems, Volume 3, Issue 4. 1997.

[Schmid99] Schmid H. Framework design by systematic generalization. Building Application Frameworks. New York: Wiley. p. 353-378. 1999.

[Tokuda01] Tokuda, L., Batory, D. Evolving Object-Oriented Designs with Refactorings. Automated Software Engineering, v. 8, p. 89-120. 2001.

[Winsen96] van Winsen P. (Re)engineering with Object-Oriented Design Patterns. Master's thesis, Utrecht University, 1996.



About the Authors

Mariela Cortés received her BSc degree from the National University of La Plata (UNLP) Argentina, MSc degree from the Militar Institute of Engineering (IME), Brazil, and a PhD degree in Computer Science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil. Her current research interests include object-oriented design and multi-agents systems development at the Software Engineering Laboratory of PUC-Rio. At present, she is adjoint professor and researcher at the State University of Ceara (UECE). Email: mariela@larces.uece.br.

Marcus Fontoura had led several framework projects and specializes in Web-based software development and service-oriented architectures in the realm of IBM's Almaden Research Center. Before that he has held research positions at the Computer Systems Group of the University of Waterloo, Canada, and at Princeton University's Computer Science Department. Email: marcusfontoura@sbcglobal.net.

Carlos Lucena received a BSc degree from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, in 1965, a MMath degree in computer science from the University of Waterloo, Canada, in 1969, and a PhD degree in computer science from the University of California at Los Angeles in 1974. He has been a full professor in the Departamento de Informatica at PUC-Rio since 1982. His current research interests include software design and formal methods in software engineering. He is member of the editorial board of the International Journal on Formal Aspects of Computing. Email: lucena@inf.puc-rio.br.