# Co-Evolving Application Code and Design Models by Exploiting Meta-Data

Walter Cazzola
Dept. of Informatics and Communication,
Università degli Studi di Milano
cazzola@dico.unimi.it

Sonia Pini
Dept. of Informatics and Computer Science,
Università degli Studi di Genova
pini@disi.unige.it

Ahmed Ghoneim, Gunter Saake
Institute für Technische und Betriebliche Informationssysteme,
Otto-von-Guericke-Universität Magdeburg
{ghoneim|saake}@iti.cs.uni-magdeburg.de

## ABSTRACT

Evolvability and adaptability are intrinsic properties of today's software applications. Unfortunately, the urgency of evolving/adapting a system often drives the developer to directly modify the application code neglecting to update its design models. Even, most of the development environments support the code refactoring without supporting the refactoring of the design information.

Refactoring, evolution and in general every change to the code should be reflected into the design models, so that these models consistently represent the application and can be used as documentation in the successive maintenance steps. *The code evolution should not evolve only the application code but also its design models.* Unfortunately, to co-evolve the application code and its design is a hard job to be carried out automatically, since there is an evident and notorious *gap* between these two representations.

We propose a new approach to code evolution (in particular to code refactoring) that supports the automatic co-evolution of the design models. The approach relies on a set of predefined meta-data that the developer should use to annotate the application code and to highlight the refactoring performed on the code. Then, these meta-data are retrieved through reflection and used to automatically and coherently update the application design models.

## Keywords

Software Evolution, Co-Evolution, Refactoring, Meta-Data, Reflection.

## 1. INTRODUCTION

Software maintenance covers most part of the software life cycle. Software maintenance takes place to tackle software flaws (*corrective maintenance*) and to adapt the application to new requirements (*software evolution* and *refactoring*). Normally, the software maintenance should follow exactly the same steps followed during the software development: starting by planning the maintenance on the application design models and then applying it on the code. This seldom happens since adapting the design models to satisfy the new requirements and then update the code takes more time than directly modify the code. Sometimes, the design models are successively modified to match the changes performed on the code.

Unfortunately, adapting the design models from the evolved code is difficult and prone to erroneous interpretations. Whereas descending through the different levels of abstraction is relatively straightforward and well supported by methods and tools, the synthesis of design information from an evolving implementation is far from obvious in particular if it should be automatically done. The development process is not a bijective function or better by reversing the implemented code we got a set of design models different from the original one. The developer implementing the design models interprets them and adopts programming techniques and patterns derived from his/her experience and skilling that cannot be replicated by any reverse engineering technique. Moreover, the design models rarely detail every aspect of the application that need the developer's cleverness to be realized. These facts contribute to form a *design/implementation gap* [3] between the two representations (design models and code) that become wider when the code has been modified hampering the detection of the changes in the code and the corresponding updating of the design models.

It is fairly evident that software maintenance and evolution impact on both the application code and design models. Since the design/implementation gap the code evolution should not precede the design models evolution but they should evolve together (*software co-evolution* [4]: software evolution requires the synchronization between different views in the software development process). At least, the gap should be filled during the code evolution by providing the necessary data to connect the evolved code to how the design models should evolve to match the evolution.

In this paper, we propose a technique, based on our previous work [2], that permits to automatically update the application design models after the code adaptation. In practice, when the developer changes the application code, he/she decorates the code with a set of pre-defined meta-data describing how the design should be adapted after the code evolution. These meta-data will be successively retrieved through reflection and used to automatically update the application design models. In this way, the developer can focus on the code evolution, saving time, and with a little effort achieving also the evolution of the application design models.

At the moment, the work focuses on the refactoring techniques (a special case of software evolution [5]) since they provide catalogs of common and well defined evolutionary situations that can be easily used as a basis for the definition of our set of meta-data. In the future, the approach will be extended to support more general software evolution activities. Analogously, we consider the UML as


*SAC* '07, March 11-15, 2007, Seoul, South Korea.

the modeling language and Java as the application programming language but the approach can be easily applied to any modeling language and programming language with meta-data support.

The rest of the paper is structured as follows: section 2, describes how meta-data can be used to trace code evolution and introduces our set of annotations. Section 3 shows on an example how to annotate the application code to trace its refactoring. Section 4 shows how the meta-data are used to update the application design models. Finally, in sections 5 and 6 we present some related work and draw our conclusions.

# 2. REFACTORING AND META-DATA

Many refactoring catalogs have been presented in the last few years. Martin Fowler in his book [5] has proposed one of the richest and most used collection of refactoring actions.

It should be well known that refactoring actions are carried out directly on the application code and that they are supported by most of the development environments. Notwithstanding that the performed changes often render obsolete and incoherent the application design, the developer performing the code refactoring rarely updates the design models as well.

A post-refactoring update of the design models is particularly difficult since the refactoring actions change the code structure breaking the link with the original design models, if any. For example, let us consider the *extract method* refactoring action, it promotes some recurring piece of code to be the body of a new method and replaces all the occurrences of that piece of code with a call to the new method. From the point of view of the design models, this impacts on the structural models by requiring the introduction of the new method and on the behavioral models (such as the sequence and the activity diagrams) by substituting the description of the *promoted* code with a call to the new method or simply by requiring the introduction of a new invocation in the actions involving the new method. After the refactoring, the code is clean and the only clue of the piece of code presence is the new method call but the removed code could crosscut many use cases and from this quite cryptic clue is not easy to go back to all these use cases. The design models update becomes still more difficult when the developer performs several refactoring actions at once.

As stated before, the urgency requirement renders difficult to coevolve the design models contemporary to the code. At the same time, to extract from the refactored code the necessary information to evolve the design model is difficult as well because there is not enough clues to reconstruct the whole history of the performed code refactoring. What it is missing is a mechanism to document the code refactoring against the design models without wasting too much time in the operation and that can be supported by the development environment. This mechanism should provide enough details and clues to automatically update the design after the code refactoring.

To this respect, we propose to decorate the code with meta-data (our clues) tracing the performed refactoring. In the case of Java, we exploit the annotation mechanism provided by the programming language. Each refactoring action differently affects the design models and specific data characterize each occurrence of the action, e.g., a *rename* refactoring action applied to a field is different than when applied to a method and the old and new names characterize each use of the action and must be reported to reconstruct it on the design models. Not only the effect of the refactoring can be scattered all around the code as in the case of the *extract method* refactoring action and the annotations must describe the role of the change, e.g., the new method and where the code has been extruded. Therefore, to each refactoring action is associated a set of annotation that permits of documenting all the aspects of the performed refactoring actions. During the code refactoring, the developer simply annotates the code with the annotation corresponding to the refactoring action he/she has performed and filling in the data identifying the specific case. The annotations do not introduce an overhead in the application execution and live inside the application since a specific tool strip them of the code and update the design models consequently.

## 2.1 Meta-Data: Java Annotations

Since Java 5 [7], the platform provides a general purpose and customizable annotation mechanism that allows the developer to define and use user-defined annotation types. The facility consists of a syntactic mechanism to declare annotations and annotate declarations, APIs to retrieve annotations through reflection, and of a class file representation for annotations.

Each new annotation requires the definition of a corresponding annotation type. Annotation types are interfaces of sorting with an extra '`@`' (at) sign before the interface definition, then the annotations can be specified in the program source by using the annotation name prefixed by an '`@`' sign.

Annotations are the corner stone of our approach to the code and design models co-evolution, but there is a significant limit into the Java annotation model, that is, its *granularity*. It is possible to annotate any kind of declarations (e.g., classes and fields declarations) but it is impossible to annotate a generic statement or code block, e.g., to put annotations inside the body of a method[1].

Code block and statement annotations are useful to trace specific aspects of some refactoring actions (e.g., to specify where the code has been removed by the *extract method* refactoring without conflicts) and to reduce the data passed to the annotation by narrowing the area affected by the refactoring action. @Java[2] is a simple extension of the Java programming language, we developed after our experience on [a]C# [1], that provides a Java 5 compliant annotation model supporting code block and statement annotations.

The meta-data are strictly coupled with the refactoring action they should trace. Depending on how a refactoring action impacts on the design models, we have to trace different information. From case to case the developer will have to use the appropriate set of annotations. For sake of brevity we cannot explain all the cases but just a couple of the refactoring actions present in the Fowler's catalog [5], further details can be read in the package documentations.

***Move Method.*** The *move method* refactoring action moves a method from a class to another to simplify the class or to decouple it from the other. The first effect on the application design is quite evident: the method changes its position and the application class diagram must reflect this fact. To post evolve the design after this kind of refactoring we must know where the method is coming from (`sourceClassName()`) and which class diagram should be changed (`classDiagramName()`).

```
@Retention(RUNTIME)
@Target(METHOD)
public @interface MoveMethod {
  String classDiagramName() default "Class Diagram";
  String sourceClassName();
}
```

The attributes of the `MoveMethod` annotation stores these data. The name of the moved method and the destination class are data that can be retrieved from the annotated element (i.e., the moved method declaration) by reflection.

[1]Note that the C# annotation model suffers of the same problem.
[2]The name @Java must be pronounced as *at-java*.

A secondary and less evident effect on the application design is represented by the changes to the calls to and from the moved method. After the refactoring, different actors are involved and sequence and activity diagrams must reflect this fact. It can be considered a secondary effect since not always the behavioral models are so detailed but in the case they have to be accordingly updated.

To this respect, we need to know which calls were shown in the design models and which ones should be shown after the refactoring. To annotate each single method call does the trick since no new method call is introduced by this refactoring action.

```
@Retention(RUNTIME)
@Target(BLOCK)[3]
public @interface MoveMethodTrail {
  String[] BehavioralDiagramNames();
  String oldTargetOID(); String oldTargetClassName();
  boolean visible() default true;
}
```

This one is a specific @Java annotation[3] and should embrace all the calls that must change in the design models. Its parameters specify the necessary data that cannot be retrieved through reflection from the code: the old target (`oldTargetOID()`) and class (`oldTargetClassName()`), in which diagram can be found (`BehavioralDiagramsNames()`) and if it must be visible or not (`visible()`).

***Extract Method.*** The *extract method* refactoring actions turns fragments of redundant code into a method whose name explains its purpose. After this refactoring action, the application structure is enriched with a new methods and the design models (i.e., the class diagram) should reflect this fact. To this respect, we need to trace which is the new method (i.e., we have just to annotate it) and which diagram is affected by the change (`classDiagramName()`). Method name, declaring class, return type, and parameter names are data that can be retrieved through reflection. In Java, the parameter names are striped from the bytecode so if we want to put them in the diagram we have to pass them to the annotation (`parameterNames()`).

```
@Retention(RUNTIME)
@Target(METHOD)
public @interface ExtractMethod {
  String classDiagramName() default "Class Diagram";
  String[] parameterNames();
}
```

Similarly to the *move method*, the *extract method* refactoring action does not exhaust its effects on the application class diagrams but it also impacts on the dynamic part of the application design models. The code extrusion leaves some gaps in the application code that are filled with an invocation to the extruded method.

```
@Retention(RUNTIME)
@Target(BLOCK)[3]
public @interface ExtractMethodTrail {
  String diagramName();
  String first(); String last();
  boolean replace() default true;
}
```

The `ExtractMethodTrail` annotation should be used to annotate those calls that should be modeled or should replace the previous modeling. The data related to the method call can be retrieved by reflection whereas the data related to the diagrams to manipulate (`diagramName()`) such as their identities and where the new action should be introduced in the diagram (`first()` and `last()`) have to be passed to the annotation.

[3]The `BLOCK` element type can be set only by using @Java, in this case the annotation is a block annotation and can be used inside a method body to embrace by braces one or more statements.


**Figure 1:** `statement()` and overprint activity diagram

## 3. META-DATA AT WORK

To show our approach in action, we use the Fowler's example (see chapter 1 in [5]). The program calculates and prints a statement of a customer's charges at a video store. The program tells which movies a customer rented and for how long then it charges them. The statement also computes frequent renter points, which vary depending on whether the film is a new release.

In its first release, the program is mainly composed of three classes: `Movie`, `Rental`, and `Customer`, whose names should be self-explanatory of what they should represent. `Customer` also has the method `statement()` that produces the statement. Figure 1 shows the code for the `statement()` method with overprint a draft of the activity diagram that describes its behavior.

The `statement()` method tends to be too long, clumsy and difficult to extend. Fowler's cuts down its complexity by performing several refactoring actions from his catalog. In our example we are just considering the first two steps:

- *extract method* | it separates the amount calculation from the method by extruding the `switch` into a new method named `getCharge()`, a rental should be passed to it;
- *move method* | `getCharge()` belongs to `Customer` but it uses information from the rental and not from the customer, the method should be moved to the right place.

Figure 2 shows the application code and design (an activity diagram overprinted on the code) after these few steps of refactoring. It also shows, emphasized by curly braces how our meta-data are used. The first refactoring action has created the method:

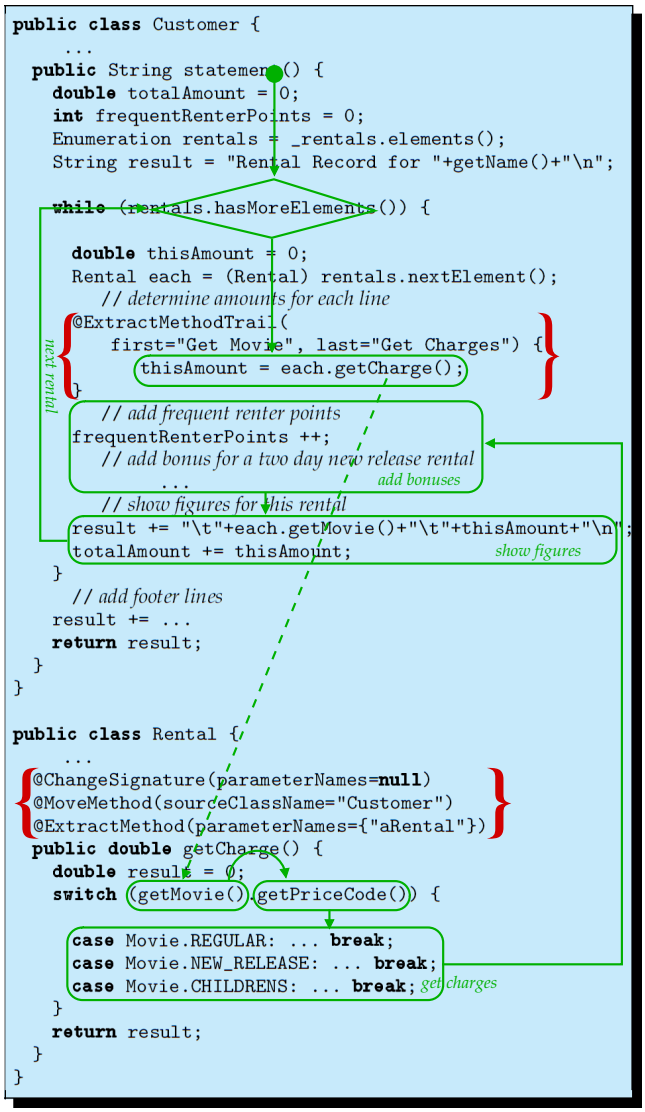


**Figure 2:** code and design after the refactoring

```
double getCharge(Rental aRental)
```

in the `Customer` class. The annotation `@ExtractMethod` traces all the necessary data that could be used to add this method to the corresponding class diagram. Whereas, the `@ExtractMethodTrail` traces the new call and the necessary data to update the corresponding diagrams (mainly the anchors in the diagram for the new action). The latter refactoring action moves the new method into the `Rental` class adjusting also the method signature. A combinations of the `@MoveMethod` and `@ChangeSignature`[4] annotations traces the change. In the same way of the refactoring actions, the annotations refer to a context that depends on the effects of the previous annotations, e.g., the `MoveMethod` annotation considers the contexts after the application of the `ExtractMethod` annotation.

From this small example, we can notice that the annotating process is up to the developer that is updating the application. The developer knows which refactoring actions has performed and the relation between the application code and design models, so he/she knows which data are relevant to successively update the design models | not all the refactoring actions must have a visible effect on the design models, e.g., we do not use the `@MoveMethodTrail` annotation even if we move a method since it is not relevant. Moreover, the developer can exploit his/her skill to reduce the amount of meta-data annotating the code even if the annotating process could be automated by an IDE supporting this process.

## 4. META-DATA AT THE RESCUE

Up till now, we have explored how to fill the gap between the refactored code and the original design model to allow a sound refactoring of the design models as well. What it is still missing is to show how the meta-data can be used to perform this task.

To this aim, we have developed a small tool, named *refactoring script generator*, that looks at the application bytecode for our annotations and collects them. This tool just exploits the Java reflection to get the annotations introduced during the code refactoring. The gathered meta-data are used to builds a jRuby[5] script that when run updates the design models after the trails left during the code refactoring. To finish the work, the tool strips all the meta-data to avoid inconsistent situations at next code/design refactoring.

Usually, the design models have a pictorical representation and UML diagrams are not an exception. This is quite important for the human understandings but it renders their automatic manipulation a hard task. To overcome this problem, the OMG provided the XMI representation for UML diagrams. In the RAMSES project[6], we have developed a Java library called *reification library* that reifies UML diagrams (i.e., their XMI representation) and permits their manipulation from Java programs. Scripting programming languages benefits when you have to generate code on-the-fly should be notorious. Moreover jRuby embeds in Ruby the possibility of interacting with Java code, characteristic that allows us of using the reification library from the jRuby scripts.

To generate a script for updating the design model rather than update them directly from the tool is a first step towards the integration of the approach with a versioning system such as CVS or SVN. The XMI files could be stored in the versioning system together with the source and the jRuby scripts provide the delta from the original and the refactored diagrams to pass from a release to another of the diagrams as well as of the code.

Due to space limits we cannot show the whole jRuby script for the example but we have to focus on the management of only one situation | the *extract method* refactoring action.

```
r = ramses.reification.Reification.new 'FowlerEx.xmi'
mycldgr = r.getClassDiagram('Class Diagram')
myclass = mycldgr.getClass('Customer')
      ...
mtd = 'getCharge'
parnames =['aRental']
partypes =[mycldgr.getClass('Rental')]
rettype = UMLType.new
myclass.addMethod(mtd, partypes, parnames, rettype)
myactd = r.getActivityDiagram('Activity diagram_1')
previous = ActivitySimpleElement
nextt = ActivitySimpleElement
newAction = myactd.addCallAction('getCharge')
nextt = myactd.getCallAction('getMovie')
previous =
   myactd.getDecisionActivityNode('for each rentals')
previous.addControlFlowTo(newAction, '', '', '')
```

[4]The `@ChangeSignature` traces the *change signature* refactoring actions since its obviousness we do not describe it further.

[5]`http://jruby.codehaus.org`

[6]`http://ramses.dico.unimi.it`

```
18 newAction.addControlFlowTo(nextt, '', '', '')
   previous.removeControlFlowTo(nextt, '')
       ...
```

The first row loads the XMI files, the filename is passed to the script generator as a parameter. Rows 2, 3, 7 and 10 reify the actors involved in the refactoring | it should not be amazing to see the names hardwired in the script since it has been generated from the available meta-data. Rows from 5 to 9 prepare, and add to the class diagram, the data related to the new method. Rows from 10 to 19 update the activity diagram after the data introduced by the `@ExtractMethodTrail` annotation. In our example, they add a new *action call* to the activity diagram for the extracted method (row 13), and connect it to the right action calls (rows 14 and 16) provided by the `@ExtractMethodTrail` annotation.

After this snippet, the script goes on dealing with the remaining meta-data. As told, the script performs the updating from the inner to the outer annotation when more then one is attached to the same code element and the outer annotations must refer to the changes performed by the inner ones. Finally, a `save()` method will reflect on file all the changes performed on UML diagrams getting the desired co-evolution.

# 5. RELATED WORK

Although it is possible to do a manual refactoring, tool support is considered crucial. Today, a wide range of tools are available that automate various aspects of software evolution and refactoring. Depending on the tool the degree of automation can vary. However, a very small set of existing tools tackle the problem of co-evolving the application design together with the code.

France et al. in [6] propose an approach to software evolution based on transformations of object-oriented models and code, called *multi-view software evolution*. Changes are performed by evolving multiple views of a system represented as models, and propagating those changes to the implementation. Propagating changes across views requires well-defined relationships among the views and the implementation defined by the programmer. This approach is not compliant to standard UML CASE tools so it can be applied only if the system adheres to this approach from the beginning.

To have more control over the software evolution, D'Hondt et al. [4] introduce the *logic-meta programming* (LMP) as a development framework to express and enforce the synchronization process. The design is expressed as a logic meta program over the implementation forcing the programmer to implement this representation. The software development process needs a new phase: after the classical design phase and implementation phase they require to translate the classical design into a logic meta-program. At this time the two levels, implementation (base-level) and new design (meta-level) are causally connected and every changes in the base level are mapped into the meta-level, but not on the classical design information losing the co-evolution.

Mens et al. in [8] propose to use intensional source code views and relations as an active documentation technique to keep application code and design synchronized. In their approach design means design documentation and the first version of the design documentation remains a largely manual process, and the application remains not coupled with design information such as UML.

Exploiting meta-data is not a new idea in the area of automatic code documentation, e.g., [9], but in general they consider only the static part of the design model (i.e., class and deployment diagrams) and only as a mechanism for self-documenting the code and not related to software evolution.

# 6. FUTURE WORKS AND CONCLUSIONS

This paper deals with the problem of co-evolving the application design models after the code refactoring. As known, the two representations are not tightly connected and it is quite hard to going back from the code to the original design models since there is a missing link between the code and the design models (*design/implementation gap*). Unfortunately, the code continuously evolves to face new requirements and its design models become soon obsolete and incoherent. The approach we have proposed and realized is based on the key idea that *the developer knows how to adapt the application design models when evolves the code*. This knowledge should be stored in the code as meta-data and successively used to evolve the design models as well without wasting time in updating them during the code evolution/refactoring.

At the moment, even we have a working prototype, the approach is in its early stages. In the future, we are going to extend it to a more general case of software evolution than code refactoring and to automate the annotation process by IDE support. Moreover, we are planning to integrate this co-evolution mechanism with a versioning system such as CVS or SVN to store the various code versions together with the corresponding design models.

## Acknowledgments

This work is part of the RAMSES project (SA 465/31-1) funded by the DFG (German Science Foundation).

# 7. REFERENCES

[1] W. Cazzola, A. Cisternino, and D. Colombo. Freely Annotating C#. *Journal of Object Technology*, 4(10):31–48, Dec. 2005.

[2] W. Cazzola, A. Ghoneim, and G. Saake. Viewpoint for Maintaining UML Models against Application Changes. In *Proc. of ICSOFT 2006*, pages 263–268, Sétubal, Portugal, Sept. 2006. Springer.

[3] W. Cazzola, S. Pini, and M. Ancona. The Role of Design Information in Software Evolution. In *Proc. of RAM-SE'05*, pp. 59–70, Glasgow, Scotland, July 2005.

[4] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-Evolution of Object-Oriented Software Design and Implementation. In *Proc. of ISSACT 2000*, pp. 207–224, Twente, The Netherlands, Jan. 2000. Kluwer.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, June 1999.

[6] R. France and J. M. Bieman. Multi-View Software Evolution: a UML-Based Framework for Evolving Object-Oriented Software. In *Proc. of ICSM 2001*, pp. 386–397, Florence, Italy, Nov. 2001. IEEE Computer Society.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, third edition, 2005.

[8] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving Code and Design Using Intensional Views - A Case Study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, July/Oct. 2006.

[9] M. Torchiano, F. Ricca, and P. Tonella. A Comparative Study on the Re-Documentation of Existing Software: Code Annotations vs. Drawing Editors. In *Proc. of ISESE'05*, pp. 277–286, Noosa Heads, Australia, Nov. 2005.