# Introducing Refactoring to Heavyweight Software Methodologies

Tiago Massoni
Electronic mail: tlm@cin.ufpe.br

**Abstract**

Refactoring is the process of changing a software system aiming at organizing the design of the source code, making the system easier to change and less error-prone, while preserving observable behavior. This concept has become popular in agile software methodologies, such as Extreme Programming (XP), which maintains source code as the only relevant software artifact. Although refactoring was originally conceived to deal with source code changes, the concept can be extended to include similar transformations on structural models of the system. Such distinction might be useful to heavyweight software methodologies, such as the Rational Unified Process (RUP), in which models are primary forms of representation. In this paper we investigate the impact of refactoring on such software methodologies, primarily addressing consistency between software artifacts after refactoring and alternatives to automatization. We also extend the Rational Unified Process (RUP) with specific activities and guidelines for dealing with refactoring.

## 1 Introduction

Among different types of program transformation, behavior-preserving source-to-source transformations are known as refactorings [26]. Refactoring changes an application aiming at improving the internal structure of the source code, yet not altering its observable behavior [9]. In this context, programmers rewrite part of the program in order to improve certain qualities, such as readability, extensibility or even aesthetics, making the system easier to change and less error-prone. Since such transformations are frequent during the overall software life-cycle, automatization is a major step towards productivity gains in software development.

Although the refactoring concept was primarily conceived to source code changes, the term can be extended in such a way that there can exist refactorings on models, since models represent the system in a higher level of abstraction. The manipulation of models may bring additional benefits to software quality and productivity, such as cheaper detection of design flaws and easy exploration of alternative design decisions. Consequently, we distinct the term "model refactoring" from "code refactoring".

One of the main reasons for the wide acceptance of refactoring as a design improvement technique is its adoption by agile software methodologies, in particular Extreme Programming (XP) [4]. XP encourages development teams to skip comprehensive initial architecture or design stages, guiding implementation activities according to user requirements and promoting successive code refactorings when inconsistencies are detected. In this approach, developers do not maintain any form of design documentation (models) in addition to source code.

However, in most heavyweight software methodologies, such as the Rational Unified Process [23], models are primary forms of representation and communication. Moreover, automatic consistency between models and source code after change is a requirement, in order to productively maintain an up-to-date abstract view of the system. This work investigates the application of model and code refactorings in such methodologies. Small examples explore refactoring across different abstraction levels, identifying potential problems and solutions. Automatization of refactorings is a major concern of this investigation. Another contribution of this work is

an extension of the Rational Unified Process with an approach for dealing with refactoring at different abstraction levels.

The paper is organized as follows. In Section 2, the refactoring concept is outlined, giving context to similar transformations on structural models. Automatization and behavior preservation for model and code refactorings are discussed. In Section 3, Extreme Programming and its support to refactoring is presented, along with a comparison with heavyweight software methodologies, in terms of model maintenance and consistency between model and source code. In Section 4, the impact of model and code refactorings at different abstraction levels is studied on small examples. Potential solutions for automatization are discussed. In Section 5, the Rational Unified Process is extended for dealing with model and code refactorings during software development and maintenance. Finally, in Section 6, the study is summarized.

## 2 Refactoring

Program restructuring [14, 2, 20] is a technique for rewriting software that may be useful either for legacy software as well as for the production of new systems. The internal structure is changed, although the behavior — what the program is supposed to do — is maintained. Restructuring re-organizes the logical structure of source code in order to improve specific attributes [20], or to make it less error-prone when future changes are introduced [2].

In the context of object-oriented development, behavior-preserving program changes are known as refactorings. The refactoring concept was introduced by Opdyke [25], yet becoming popular by Fowler's work [9] and Extreme Programming (XP) [4], an agile software development methodology. According to Fowler, refactoring is the process of changing a software system in such a way that it does not alter the observable behavior of the source code, yet improving its internal structure [9]. In this context, a refactoring is usually composed of a set of small and atomic refactorings (the mechanics), after which the target source code is better than the original with respect to particular quality attributes, such as readability and modularity.

Refactoring can be viewed as a technique for software evolution throughout software development and maintenance. Software evolution can be classified into the following types [19]:

- Corrective evolution: correction of errors;
- Adaptive evolution: modifications to accommodate requirement changes;
- Perfective evolution: modifications to enhance existing features.

Refactoring is mostly applied in perfective software evolution, although it also affects corrective and adaptive evolution. First, well-organized and flexible software allows one to quickly isolate and correct errors. Second, such software ensures that new functionality can be easily added to address changing user requirements.

A known issue about refactorings is automatization. Small steps of refactoring have usually been performed manually, or with primitive tools such as text editors with search and replace functionality. This situation eventually leads to corrupt design of the source code (software entropy [18]), mostly due to the fact that manual refactoring is tedious and susceptible to errors [26]. Although the choice of which refactoring to apply is naturally made by human, automatic execution of refactorings might result in a major improvement in productivity. For instance, current software development environments (e.g. Eclipse [8]) include a predefined set of refactorings that can be automatically applied to programs.

In addition, concerning behavior preservation, XP informally guides refactoring assisted by unit tests, increasing confidence in the correctness of a sequence of transformations. However, the correctness of the transformation steps in the sense that they preserve behavior is not formally justified. Furthermore, verification of object-oriented programs is highly nontrivial. A

number of recent research initiatives have pointed out directions for formally justifying refactorings. In Opdyke's work, preconditions for refactorings are analyzed [25], whereas Robert's work formalizes the effect of refactorings in terms of postconditions, in order to build efficient refactoring tools [26]. In contrast, Mens et al. [24] apply graph representation to aspects that should be preserved, and graph rewriting rules as a formal specification for refactorings.

## 2.1 Model Refactoring

Although refactoring — or, more generally, restructuring — is originally conceived to change programs, the concept can be extended to address similar structural transformations on models of a system. Model refactoring can be defined as model transformations that improve specific qualities of the model, such as extensibility, making the perfective evolution task more manageable. In the remainder of this paper, we differentiate the terms "model refactoring" and "code refactoring".

France et al. [10] identify two classes of model transformations: vertical and horizontal transformation. Vertical transformations change the level of abstraction, whereas horizontal transformations maintain the level of abstraction of the target model. A model refactoring is an example of horizontal transformation. In contrast, the Model-Driven Architecture (MDA) approach [28], in which abstract models automatically derive implementation-specific models and source code, provides examples of vertical transformations.

Similarly, Porres [16] define mapping and update transformations. While a mapping transformation is similar to a refinement relationship, defining traceability relationships between model elements at different abstraction levels, an update transformation adds, deletes and updates elements at the same level, where model refactoring fits best.

Applying refactoring to models rather than to source code can encompass a number of benefits [10]. First, software developers can simplify design evolution and maintenance, since the need for structural changes can be more easily identified and addressed on a abstract view of the system. Second, developers are able to address deficiencies uncovered by model evaluation, improving specific quality attributes directly on the model. Third, a designer can explore alternative decision paths in a cheaper way (although small prototypes may be necessary). An apparent scenario for model refactorings is the incorporation of design patterns into a system's design model [21].

As the idea of refactoring models adds simplicity to software evolution, automatization and behavior preservation are even more complex issues when dealing with models. Editing a class diagram may be as simple as adding a new line when introducing an association, but such changes must include identifying lines of affected source code, manually updating the source, testing the changes, fixing bugs and retesting the application until the original behavior is recovered [31]. Methods and tools for partially or even totally removing human interaction in this process are invaluable for the refactoring practice.

Likewise, checking whether behavior is preserved after a model refactoring is complex, since a formal description of behavior in abstract models is rarely provided, in contrast to source code. In addition, constraints are hard to identify and inconsistencies are not easily caught by modeling tools. The most common approach is to break a high-level model refactoring into small low-level model refactorings. If semantic properties hold during each small transformation, the composite refactoring is considered to be behavior-preserving. As an example of this approach, Sunyé et al. [30] argue that behavior-preserving properties of a basic UML model refactorings can be guaranteed based on the Object Constraint Language (OCL) [22] formulas at the UML meta-model level.

# 3 Software Methodologies and Refactoring

Regarding the adoption of refactoring by modern software methodologies, Extreme Programming (XP) [4] and agile methodologies are the most visible examples. In this section, we describe the support for refactoring in XP and examine aspects of heavyweight software methodologies that will affect refactoring.

## 3.1 Extreme Programming and Refactoring

One of the main reasons for the wide acceptance of refactoring as a design improvement technique is its adoption by Extreme Programming (XP). In XP, refactorings are applied in specific parts of the code that contain inconsistencies ("code smells"), and unit tests check the software output after structural changes. In particular, besides managing software evolution, refactoring is intrinsic to each XP development activity. XP practices guide simple implementation according to immediate user requirements, promoting successive refactorings for improving design before adding new features.

In addition, despite of including modeling activities in early stages of development, XP maintains source code as the only relevant software artifact. Accordingly, model artifacts are disposed and consistency between models and source code is not considered. Therefore, model refactoring and its implications are not applicable to XP and similar methodologies.

## 3.2 Heavyweight Software Methodologies and Refactoring

OPEN [13] and the Rational Unified Process (RUP) [23] are examples of heavyweight methodologies. RUP, the most popular, is focused on visual modeling (UML [5]) and use-case driven, architecture-centric, iterative software development.

In such a methodology, defining a stable architecture baseline promotes the search for reuse opportunities and flexibility since early life-cycle iterations. In this context, model documents are primary forms of software specification and team communication. As a consequence, model refactoring may be applied on analysis and design activities, in order to obtain the benefits mentioned in Section 2.1.

Different modeling approaches may be taken, varying the abstraction level, which affect the refactoring process. One is to model an abstract view of the system, providing a clearer and more flexible description of elements and their relationships. An alternative is to produce models that are closer to source code, representing the design of an specific implementation; the designer is constrained to use a subset of the modeling language that fits the implementation's constraints [15]. In RUP, the first approach is suitable for analysis models, whereas the latter is appropriate for design models.

Furthermore, RUP requires traceability, i.e. the ability to control the consistency between software artifacts produced during different stages of the life-cycle. In this context, updates to models must be reflected to the source code, whereas updates to source code affect the corresponding models, clearly influencing refactoring. Moreover, maintaining consistency between model and source code is often a tedious and error-prone task, requiring tool support. In Section 4 we consider automatization aspects when investigating refactoring across different levels of abstraction.

Different model-code consistency approaches have been commonly used in software development, such as:

- **Simple forward engineering**. Models are produced solely in early stages of development. Once a skeleton of the code is generated, models are discarded and all changes will be made directly to code. Consistency effort is not required, yet the value of abstract views of the system is lost. This is a common approach in XP projects.

- **Successive reverse engineering**. Source code is still the primary artifact, but for each development iteration, models are generated as physical images of the source code. Although the consistency cost is low — reverse engineering tools are widely available — models are significantly close to code, obscuring characteristics that are related to the problem being modeled.
- **Round-trip engineering**. Based on initial models, skeleton source code is generated, being completed during implementation activities. When developers have a stable version of the source code, reverse engineering reconstructs as best as possible an up-to-date version of models, restarting the cycle when dealing with changes. The process is strongly dependent on tool support.

Figure 1 distributes these approaches across levels of model importance, ranging from code-centered development to complete model-driven development (the Model-Driven Architecture (MDA) approach).

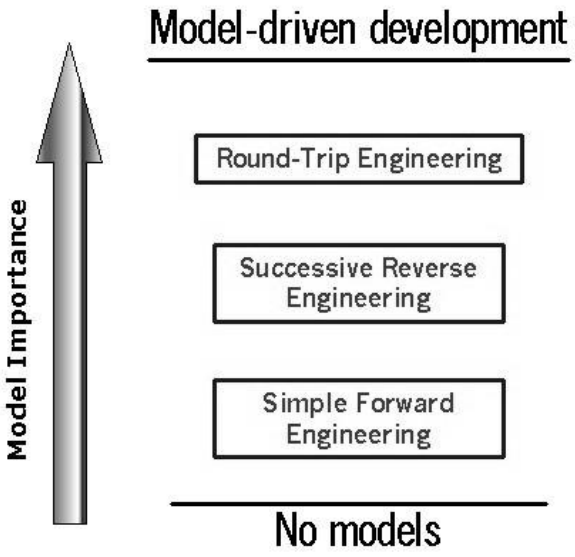


Figure 1: Model-code Consistency Approaches.

# 4 Relationship Between Model and Code Refactoring

In this Section we show small examples of model and code refactorings across different levels of abstraction. We try to identify potential problems and possible solutions by examining the impact of refactorings on model-code consistency, highly desirable in RUP and other software methodologies.

## 4.1 Refactorings

Our refactorings are applied on classes of a simple banking application. The examples are presented as UML class diagrams for modeling structural elements and Java [12] source code. Behavioral models are surely affected by our refactorings, although not considered here. The affected classes are represented in three abstraction levels: analysis model (implementation-independent), design model (implementation-dependent) and Java source code. While analysis models outline an abstract view, design model and source code realize those classes as Enterprise Java Beans, on the J2EE platform [29].

These examples are based on Fowler's refactorings [9] and design pattern introduction. In order to simplify the discussion, we do not consider inheritance. The refactorings are composed of primitive refactorings (steps), which are loosely based on the refactorings introduced by Roberts [26]. Essentially, we are interested in how model and code refactorings (and their semantic constraints) relate across abstraction levels, including automatic transfer of changes by round-trip engineering. For each example, we consider three change scenarios: (1) refactoring on the source code, (2) refactoring on the concrete model and (3) refactoring on the abstract model, as illustrated in Figure 2.

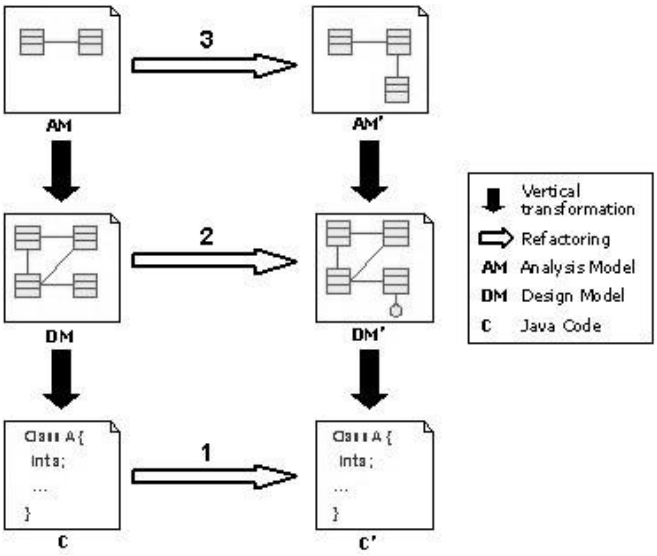


Figure 2: Refactoring Scenarios.

### 4.1.1 Extract Class

This refactoring is applied when the behavior of a class is more appropriate distributed into two classes, promoting reuse and extensibility. The mechanics consists in creating a new class and moving the relevant fields and methods from the old class into the new class, as depicted in Figure 3. Table 1 shows sample decompositions of Extract Class into primitive refactorings for each scenario, along with side changes required on each primitive refactoring for preserving program's behavior.

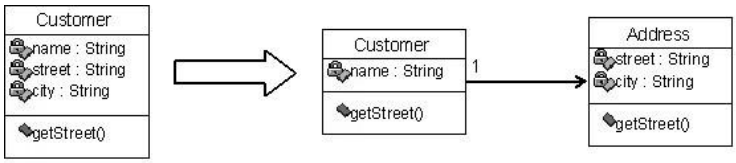


Figure 3: Extract Class.

**Code Refactoring.** The primitive code refactorings are listed in Table 1. Particularly, some of these steps will be extended in order to handle implementation-specific changes. In Extract Class, the developer has to decide how to expose the newly created Address class. If Address is more than a simple immutable value object, it must be implemented as a enterprise bean, i.e. the class must implement the EntityBean interface and respective home and remote interfaces must be created (optional steps in Table 1).

| Java code | Design Model | Analysis Model |
|---|---|---|
| 1. Add empty class Address, (optional) implementing EntityBean<br>2. Add instance variable address to Customer, referencing one instance of Address<br>3. Move instance variables street and city to Address<br>• Accessor methods are created in Address<br>• All references to street and city within Customer must be replaced with calls to their accessor methods in Address<br>4. (optional) Create home and remote EJB interfaces for Address | 1. Add empty class Address<br>2. (optional) Add realization Address to EntityBean<br>3. Add 1:1 association from Customer to Address<br>4. Move instance variables street and city to Address<br>• Accessor methods are created in Address<br>5. (optional) Create EJB home and remote interfaces for Address | 1. Add empty class Address<br>2. Add 1:1 association from Customer to Address<br>3. Move instance variables street and city to Address |

Table 1: Extract Class Steps.

Assuming that Extract Class is first applied on the source code, the transfer of this refactoring to the design model is expected to be easily automated. Reverse engineering tools can update the design model appropriately, which directly reflects source code constructs, although some information may be lost by round-trip engineering (e.g. source code comments). In such tools, source code with new Address class implementing EntityBean is supposed to generate a model with classes and the correct realization relationship, as well as new address attribute in Customer must generate the proper 1:1 association relationship.

On the other hand, updating analysis models from code is harder, since we need to abstract from implementation details that do not apply to the model. In this example, EJB interfaces and realization relationships are not relevant to the analysis model, hence related transformations do not map. This process is not straightforward for reverse-engineering tools, requiring controlled traceability between abstract and concrete elements. Nevertheless, the generation of an intermediate design model during the process certainly simplifies the task.

**Design Model Refactoring.** There is more to model refactoring than only adding and moving model constructs in UML CASE tools. In fact, it is hard to measure the impact of a simple refactoring on the whole model, since semantic properties are not easily extracted from a UML model and changes cannot be validated by testing. Additionally, a minor model transformation can affect a major portion of the source code, depending on the abstraction level of the model.

Assuming that Extract Class is applied on the design model, the mapping to analysis model is similar to the previous scenario. Only changes affecting relevant entities will map (e.g. adding Address class and its association relationship with Customer), thereby requiring tools that record how design elements trace to analysis elements.

On the other hand, mapping model transformation to source code is not as simple. Whereas adding Address class implementing EntityBean can be automatically transferred to steps 1 and 2 in the source code, side changes of moving street and city variables (references update) cannot be captured by code generation. Therefore, additional work is required for completing the refactoring on source code, which is done manually.

**Analysis Model Refactoring.** The impact of analysis model refactoring on lower abstraction levels is even more unpredictable, compared to design models. Changes in these models may provoke a large set of design model and source code transformations, depending on the chosen

platform.

Assuming that Extract Class is applied on the analysis model, transferring changes to the design model is an elaborate task. In this example, creating the Address class on analysis may imply that the class implements EntityBean and spawns the creation of EJB home and remote interfaces on design. Code generation tools cannot avoid additional manual work for completing the refactoring.

Moreover, mapping changes on analysis model to source code is even more complex. Moving instance variables street and city to the new class maps to several modifications on code, in order to preserve behavior. Complementary manual work is required. Again, the task can be simplified by the generation of an intermediate design model.

### 4.1.2 Hide Delegate

This refactoring is applied when a client calls methods of another class' field. In this context, the client class is aware of details about the delegate object, then not following encapsulation principles. In Figure 4, we show the application of Hide Delegate on a banking application. Table 2 shows the scenarios of Hide Delegate along with side changes.

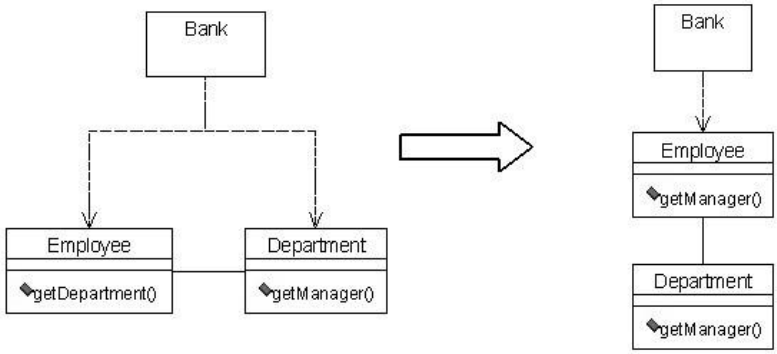


Figure 4: Hide Delegate.

| Java code | Design Model | Analysis Model |
|---|---|---|
| 1. Add method getManager() to Employee, which calls the delegate method in Department. Employee's EJB remote interface must be updated<br>2. Remove method getDepartment() from Employee. Update Employee's EJB remote interface<br>• In Bank, replace all references to Department methods by references to Employee's methods, such as getManager()<br>• In Bank, remove all calls to the getDepartment() method | 1. Add method getManager() to Employee. Employee's EJB remote interface must be updated<br>2. Remove method getDepartment() from Employee and update Employee's EJB remote interface<br>• Remove dependency relationship between Bank and Department | 1. Add method getManager() to Employee<br>2. Remove method getDepartment() from Employee<br>• Remove dependency relationship between Bank and Department |

Table 2: Hide Delegate Steps.

**Code Refactoring.** Assuming that Hide Delegate is first applied on source code, the changes will be transferred to the corresponding design model. In this situation, reverse engineering

tools present satisfactory results, including EJB updates. However, we may experience some problems, specially with the dependency relationship, which is not updated correctly in most UML CASE tools. For instance, side changes when removing getDepartment() from Employee in the source code do not transfer to the model.

Concerning analysis models, the transfer is even more problematic, although an intermediate design model certainly simplifies the task. Reverse engineering tools work for the most relevant model elements, yet dependency relationship and abstraction of EJB details are more complex issues.

**Design Model Refactoring.** If Hide Delegate is applied to the design model, transferring the steps to the analysis model is defined by the ability of abstracting EJB-related changes and non-relevant design elements.

Concerning the transfer of design model transformations to source code, forward engineering tools can generate a skeleton of the new method getManager() in the Employee class, but programmers must add the delegation code manually. In addition, removing Bank—Department dependency relationship does not translate to the corresponding side changes on source code (they also must be performed manually). However, updates to EJB interfaces can be transferred directly.

**Analysis Model Refactoring.** Assuming that Extract Class is applied to the analysis model, corresponding design model changes are not easily predicted. In this example, changes to EJB interfaces can only be transferred if implementation information is used to generate lower-level changes. If new and removed methods (getManager() and getDepartment()) are relevant to the analysis, then these changes should be straightforward.

However, transferring changes to source code is even harder, due to the significant level difference. For example, implementation-related side changes and EJB specifics are impossible to infer from abstract refactoring. An intermediate design model is highly desirable in the transfer process.

### 4.1.3 Strategy Pattern Introduction

One of the most usual applications of refactoring is to introduce design patterns [11, 6]. For example, we can apply the Strategy pattern to an object that may contain different ways to perform a specific operation. A new class is created to represent a particular algorithm, being referenced by the original class. In Figure 5, we show the introduction of this pattern to our banking application, which modularizes a strategy to calculate the amount of state tax for each deposit to a checking account. In this example, we consider a limited version of the Strategy pattern, without inheritance, for simplifying our discussion. Table 3 shows the refactoring steps along with side changes, for each scenario.

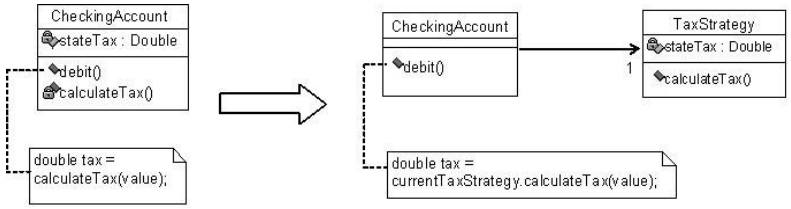


Figure 5: Strategy Pattern.

**Code Refactoring.** Assuming that Strategy pattern is applied to the source code, reverse engineering can transfer all the changes to the corresponding design model. In this case, we

| Java code | Design Model | Analysis Model |
|---|---|---|
| 1. Add empty class TaxStrategy<br>2. Add instance variable currentTaxStrategy to CheckingAccount, referencing one instance of TaxStrategy<br>3. Move instance variable stateTax to TaxStrategy<br>• Accessor methods are created in TaxStrategy<br>• All references to stateTax within CheckingAccount must be replaced with calls to its accessor methods in TaxStrategy<br>4. Move method calculateTax to TaxStrategy<br>• Either all references to calculateTax must be updated or a delegation method is maintained in CheckingAccount | 1. Add empty class TaxStrategy<br>2. Add 1:1 association from CheckingAccount to TaxStrategy<br>3. Move instance variable stateTax to TaxStrategy<br>• Accessor methods are created in TaxStrategy<br>4. Move method calculateTax to TaxStrategy | 1. Add empty class TaxStrategy<br>2. Add 1:1 association from CheckingAccount to TaxStrategy<br>3. Move instance variable stateTax to TaxStrategy<br>4. Move method calculateTax to TaxStrategy |

Table 3: Strategy Pattern Steps.

assume that EJB home and remote interfaces are not affected by the refactoring, since calculateTax is a private method (not published in interfaces) and TaxStrategy do not change the EJB implementation.

Similarly, the relevant analysis elements affected by the refactoring can be easily updated from source code, since EJB interfaces are not affected in this example.

**Design Model Refactoring.** If Strategy pattern is applied to the design model, design entities that are relevant to analysis may be easily updated, since EJB aspects are not affected.

On the other hand, updating the source code by forward engineering is not as straightforward. Side changes in steps 3 and 4 have to be completed manually. Furthermore, moving methods in the model brings additional problems to round-trip engineering in most UML CASE tools. For instance, if we move calculateTax() to TaxStrategy in the model, the generated source code contains a skeleton implementation of calculateTax() in TaxStrategy whereas the body of the method is lost.

**Analysis Model Refactoring.** Applying the Strategy pattern to the analysis model, the design model will lack accessor methods for stateTax in TaxStrategy, although implementation elements are not affected in the transformation.

Concerning the source code, side changes are performed manually after forward engineering (with intermediate design model). However, relevant analysis elements are updated accordingly in code.

### 4.2 Alternative Solutions for Automatization

From these small examples, we can identify a number of problems concerning round-trip engineering and consistency between models and source code after refactoring. Even though our examples do not represent all possible situations in practice, we believe that problems occur on most refactoring scenarios, using UML CASE tools and round-trip engineering.

Most problems are found when generating source code from transformed models, besides

difficulties in dealing with different abstraction levels and implementation specifics. In this context, we can try alternatives for automatic consistency between model and code refactoring.

**Attaching source code to models.** In this solution, a CASE tool would keep structural model elements attached with respective source code. Therefore, model refactoring steps are extended with source code transformations, specially side changes that otherwise would be manually performed. As an advantage, applying refactoring to models does not require additional manual changes to source code. On the other hand, developers lose abstraction capability, since models must be concrete enough to include code constructs. For instance, analysis models would hardly be maintained by such a tool, since it may be impossible to abstract from implementation specifics.

**Executable UML.** This solution encompasses other recently-devised concepts, such as MDA [28] and precise action semantics [1]. The idea is that, rather than elaborate an analysis model into a design model and then produce code, application developers use tools to translate abstract application constructs, annotated with action semantics and Object Constraint Language (OCL) [22] tags, into executable entities [3]. In this context, refactorings on abstract models are smoothly transferred to source code. However, this approach requires high-quality model compilers and Virtual Execution Environments (VEEs), providing infrastructure upon which executable UML models can execute. In fact, these tools are in a very incipient stage, and the applicability of this technique is an open research area.

**Recording refactorings.** This solution defines an approach to automatic refactorings across different abstraction levels, yet not applying round-trip engineering. By this approach, refactoring steps on software artifacts (models or source code) at a source abstraction level are recorded by a CASE tool, followed by the mapping of those steps to a target level. Finally, the mapped steps are applied to artifacts at the target level. For example, steps for a model refactoring can be recorded and applied to source code. As a result, developers can minimize manual refactoring, eliminating the need for code generation and all its associated problems.

However, there is a number of open questions with this approach. First, it is hard to check whether mapped transformations preserve behavior across abstraction levels. Provided we split refactorings into primitive refactorings (or steps), we can map each step to corresponding target steps. Consequently, we can assure that if these steps preserve behavior in isolation, the composite refactoring will be behavior-preserving. Still, the challenge is to define a minimal set of primitive refactorings for each abstraction level and elaborate a mapping mechanism between levels.

Second, mapping steps on analysis models to design models, or vice-versa, requires traceability information from abstract to concrete level. Otherwise, the tool will not be able to map abstract changes to concrete changes, which usually affect implementation-specific entities that are not present in analysis.

Third, side changes, which are usually done before each step in order to preserve behavior, are not easily predicted when mapping abstract to concrete changes. However, we believe that side changes are closely related to refactorings preconditions [25]. A program (or model) must satisfy a number of preconditions before each primitive refactoring is applied, for certifying that the behavior is preserved. Accordingly, the tool may be able to deduct the correct side changes for a step by analyzing its preconditions and the program's (or model's) current state. Otherwise, manual adjustments will be required, mainly for source code.

Figure 6(a) shows horizontal and vertical transformations on this approach, which can be considered a mid-term between the XP approach to model consistency (Figure 6(b)) and formal methods approach (Figure 6(b)). In XP, models are initially used but discarded after code

generation, while in a formal approach refactorings would be applied to models (specifications) and models would evolve to source code by precise refinement.

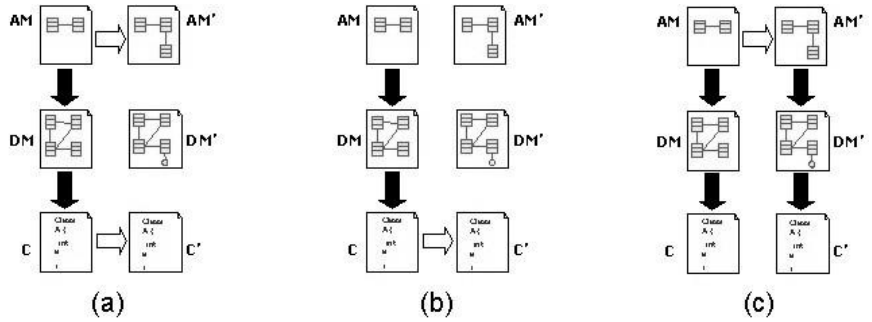


Figure 6: Approaches to Model-Code Consistency.

# 5 Extending The Rational Unified Process (RUP) With Refactoring

In this Section we present suggestions for extending RUP with support for planning, applying, and evaluating model and code refactoring. We believe this extension might be useful since refactoring is an activity that may accompany most stages of the software development life-cycle. For instance, even attempts to understand foreign code can motivate simple refactorings (e.g. by adding comments). Some of our suggestions are based on the work of France et al [10], which introduces a framework for software processes dealing with model evolution.

During software process enactment, refactoring can be supported by the following basic tasks:

- Planning and evaluation of quality improvements;
- Decision-making about where to refactor and what refactoring to apply;
- Application of model and code refactoring;
- Writing unit tests prior to implementation in order to support code refactoring.

Based on these basic tasks, we point out parts of RUP that may be changed for dealing with refactoring in a software process. First, we outline possible changes to static aspects of RUP (activities, steps, artifacts). Next, we outline how dynamic aspects (phases and iterations) can be affected by refactoring support.

## 5.1 Suggestions for Static Aspects

Changes for introducing support for refactoring will be located in the following RUP workflows.

**Project Management.** The task of planning and evaluating refactoring is more suitable for iteration planning, since it is simpler to determine quality improvements focusing on smaller project goals (e.g. a set of use cases). In this workflow, we extend the workflow detail *Plan for The Next Iteration* with new activities for planning and evaluating refactoring, as showed in Figure 7. These activities will produce and update a new artifact, the Refactoring Plan.

For setting refactoring goals, we suggest a new activity, Develop Refactoring Plan. This activity derives a set of questions (high-level goals), whose answers can define how these goals are satisfied. Based on these goals, next steps can establish metrics that provide a practical

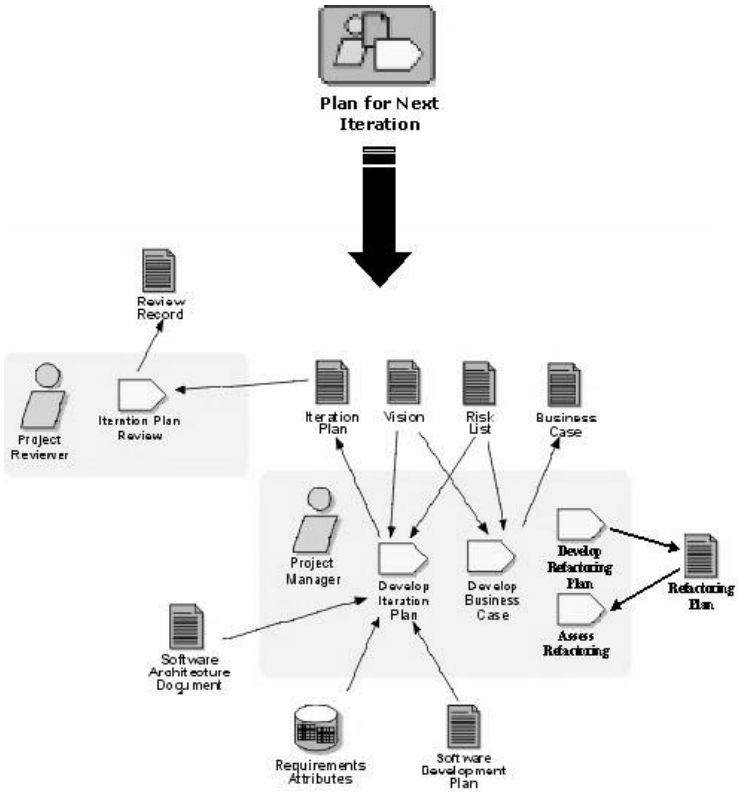


Figure 7: Suggestions for Planning Refactorings.

way to identify "bad smells", i.e. properties of software artifacts that may require quality improvements. As examples of such metrics, average number of attributes and methods in individual classes may help decide to apply the Extract Class refactoring, whereas a low number of references to a class may lead to the Inline Class refactoring. Accordingly, relevant research work has been done on this topic [7, 27].

In addition, the activity Assess Refactoring evaluates the updated models and source code, based on the evaluation criteria developed in the activity Planning Refactoring (high-level and metrics). The output of this activity is certainly useful for future iterations. Also, the manager must check whether refactoring provides the expected return of investment, since developers are restructuring code that was previously working.

**Configuration and Change Management.** Even though we do not suggest any changes to this workflow, its activities are closely related to refactorings. Configuration management tools are required for keeping track of different versions of artifacts during refactoring. In addition, complex refactorings, particularly on models, may require submitting formal change requests for management control.

**Analysis and Design.** In this workflow, architects and designers must take technical decisions concerning the application of refactoring on models. Next, model refactorings are applied. We identify existing activities that might receive additions for achieving these goals.

Architects and designers must investigate issues and find new opportunities for reuse in design models (and analysis models, if maintained). These steps are based on the metrics defined in the Develop Refactoring Plan activity, as they are applied to models in order to check the need for quality improvements. Steps for addressing these needs can be included to Review Architecture

and Review Design activities (Refine Architecture and Design Components workflow details, respectively). Furthermore, the architect or designer must decide whether to do model or source code refactoring. Usually, model refactorings are more useful for major structural changes (e.g. introduction of one or more design patterns) or exploring alternative designs.

For the identified refactorings, we suggest new steps for their application to model artifacts. These steps surely affect activities contained in the following workflow details, depicted in Figure 8:

- **Define a Candidate Architecture**, affecting activities Architectural Analysis and Use Case Analysis;
- **Refine the Architecture**, affecting activities Incorporate Existing Design Elements;
- **Design Components**, affecting activities Use-Case Design, Subsystem Design and Class Design;
- **Design the Database**, affecting activity Database Design.

The affected activities include steps for applying the refactorings and updating other model artifacts which contain changed elements. For instance, a refactoring that adds a new class to the system will have impact on use-case and database design. In addition, these refactoring steps may also be performed for mapping code refactorings to models, using approaches described in Section 4. The impact of these changes on the source code are considered by implementation activities.

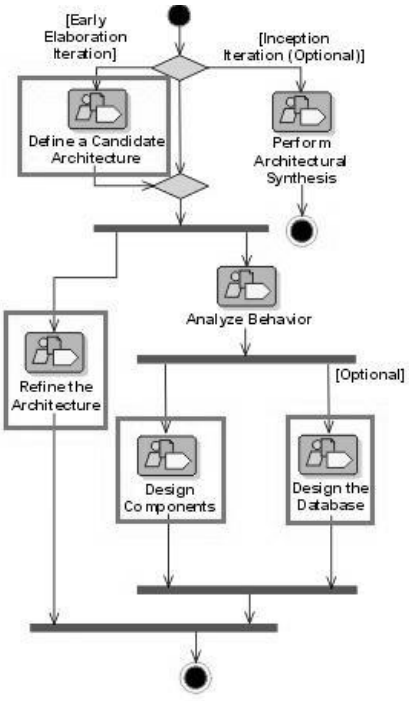


Figure 8: Analysis and Design Workflow Details Affected by Refactoring.

**Implementation.** In this workflow, implementers carry out code refactoring, upon deciding when it is the case. We suggest new steps to be included to existing activities.

Initially, refactorings applied to analysis and design models must be reflected to code. The activity Structure the Implementation Model covers this need, usually by forward engineering

in existing CASE tools, although, as seen in Section 4, this technique does not reflect changes completely.

Furthermore, a new step must be added in order to identify points in the source code that may need refactoring. Small changes, usually below method level, are candidate code refactorings. This decision is also based on metrics defined in the Refactoring Plan. Review Code (Implement Components workflow detail) is the most appropriate activity to contain this new step.

After deciding when and where to refactor source code, apply the refactorings as a new step in the Implement Component activity, supported by tests previously written (see Test workflow). In this step, automatic application of code refactoring is very useful for increasing productivity and avoiding errors. Furthermore, implementers must provide feedback of these changes to design models, which are also be performed during this activity.

**Test.** Previously-written unit tests can help maintaining the observable behavior of programs after code refactoring. This practice has widespread acceptance in the XP community, as a complementary technique for refactoring programs. In this context, activities for implementing and executing tests and the step for applying code refactoring must be combined, in order to provide an efficient test support for refactoring.

### 5.2 Suggestions for Dynamic Aspects

Considering dynamic aspects of RUP, we outline how refactoring can be performed in iterations of each RUP phase. Since the intent of the first phase (Inception) is to make the business case needed to justify launching the project [17], we do not consider refactoring in its iterations.

Concerning the remaining phases, starting a metaphor and evolving the design only by merciless refactoring as in XP does not fit into RUP. However, refactoring can be useful for evolving the baseline architecture during elaboration and for corrective, adaptive and perfective changes during construction phase. We discuss support for refactoring in those phases.

**Elaboration.** In this phase, the architecture baseline is developed and validated, becoming the basis for all following development activities. For establishing a stable architecture, refactoring is a valuable technique, since changes are constant until the architect achieves the best foundation for the system to be built. In addition, due to the fact that major changes and exploration of alternative designs are commonly seen in this phase, model refactoring may be more appropriate, but, of course, keeping consistency with the associated source code.

**Construction.** In this phase, a software product ready for initial operation in the user environment is developed, based on the architecture baseline. We believe that, since the architecture is considered stable, major refactorings will be minimal. However, requirement changes are important risks to the project, and refactoring is relevant before adding new features. Code refactoring is often applied during implementation activities, reflecting structural changes to models. Nevertheless, model refactoring can be useful when a higher-level view of the system is needed to perform changes.

**Transition.** This phase focus on establishing the product in the operational environment, handling all the issues needed for operation in the user environment. Although most core workflows of RUP (analysis and design, implementation and test) play a less significant role during this phase, model and code refactoring may still be necessary for helping correct defects and addressing unforeseen problems.

## 6 Conclusion

In this paper, we investigated the role of model and code refactorings in heavyweight software methodologies, such as RUP. We studied the impact of refactoring on consistency between model and source code, required in such methodologies, and alternatives to automatization. Also, we suggested an extension of RUP supporting refactoring in its workflows and phases.

The results of our investigation can be used in the development of new CASE tools for increasing productivity in the refactoring practice, specially in the relationship between model and source code transformations. Moreover, our extension of RUP might be used as a basis for process improvement, aiming at better software evolution support.

In our point of view, recording refactorings is the most promising approach for dealing with model and code refactorings, although having many open issues. For instance, behavior preservation and the relationship between abstract and concrete constructs are critical for the application of this approach in RUP and other software methodologies. We believe that initiatives for formalizing refactorings with less ambiguous notations point out research directions on these topics.

## References


[1] Action Semantics Consortium. Precise Action Semantics for the Unified Modelling Language, 2000. http://www.kabira.com/as/.

[2] Robert S. Arnold. Software Restructuring. *Proceedings of the IEEE*, 77(4):607–617, April 1989.

[3] Marc J. Balcer and Stephen J. Mellor. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[4] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[5] Grady Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison-Wesley, 1999.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[7] Serge Demeyer, Stphane Ducasse, and Oscar Nierstrasz. Finding Refactorings via Change Metrics. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 166–177. ACM Press, 2000.

[8] Eclipse.org. Eclipse Project, 2003. http://www.eclipse.org.

[9] Martin Fowler. *Refactoring—Improving the design of existing code*. Addison Wesley, 1999.

[10] Robert France and James M. Bieman. Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In *Proceedings of The IEEE International Conference on Software Maintenance*, November 2001.

[11] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] James Gosgling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[13] I. Graham, B. Henderson-Sellers, et al. *The OPEN Process Specification.* ACM Press, Addison-Wesley, 1997.

[14] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, University of Washington, 1991.

[15] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping UML Designs to Java. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 178–187. ACM Press, 2000.

[16] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. Technical report, Turku Center for Computer Science, April 2003.

[17] Ivar Jacobson et al. *The Unified Software Development Process.* Addison-Wesley, 1999.

[18] Frederick P. Brooks Jr. *The Mythical Man-Month (Anniversary Ed.).* Addison-Wesley Longman Publishing Co., Inc., 1995.

[19] Sheena R. Judson, Doris L. Carver, and Robert France. A Metamodeling Approach to Model Refactoring. Submitted to UML' 2003.

[20] B.-K. Kang and J. M. Bieman. A Quantitive Framework for Software Restructuring. *Journal of Software Maintenance*, 11:245–284, 1999.

[21] Joshua Kerievsky. Refactoring To Patterns. Draft available at http://industriallogic.com/-papers/rtp017.pdf.

[22] Anneke Kleppe and Jos Warmer. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[23] Philippe Kruchten. *Rational Unified Process - An Introduction.* Addison-Wesley, 1999.

[24] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising Behaviour Preserving Program Transformations. In *Proceedings of the First International Conference on Graph Transformation*, pages 286–301. Springer-Verlag, 2002.

[25] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[26] Don Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[27] Frank Simon, Frank Steinbrckner, and Claus Lewerentz. Metrics Based Refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, 2001.

[28] Richard Soley. Model Driven Architecture, November 2000. OMG Document omg/2000-11-05.

[29] Sun Microsystems. Java 2 Enterprise Edition Specification, 2003. Available at http://-java.sun.com/j2ee/.

[30] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML Models. In *The Unified Modeling Language 4th International Conference*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

[31] Lance Tokuda and Don S. Batory. Evolving Object-oriented Designs with Refactorings. In *Automated Software Engineering*, pages 89–120, 1999.