

# Cascaded Refactoring for Framework Evolution

Greg Butler  
Department of Computer Science  
Concordia University  
Montreal, Quebec, H3G 1M8 Canada  
gregb@cs.concordia.ca

Lugang Xu  
Department of Computer Science  
Concordia University  
Montreal, Quebec, H3G 1M8 Canada  
lxu@cs.concordia.ca

## ABSTRACT

Refactoring of source code has been studied as a preliminary step in the evolution of object-oriented software. We extend the concept of refactoring to the whole range of models used to describe a framework in our methodology: feature model, use case model, architecture, design, and code. We view framework evolution as a two-step process: refactoring and extension. The refactoring step is a set of refactorings, one for each model, that cascades through them. The refactorings chosen for a model become the rationale or constraints for the choice of refactorings of the next model.

The cascading of refactorings is aided by the alignment of the models. Alignment is a traceable mapping between models that preserves the commonality-variability aspects of the models.

## Keywords

framework, evolution, refactoring, domain analysis, use case, feature model, architecture, hotspot, design pattern, role

## 1. INTRODUCTION

Systematic reuse using domain engineering, product lines, or frameworks has been successful in delivering the economic benefits of reuse. An object-oriented framework is a concrete realization in source code of a domain-specific software architecture. While frameworks are often cited as an example of the reuse of requirements, architecture, design, and code, there is a prevailing conception that the only concrete model used for frameworks is the source code itself.

Methodologies for the development of a framework have been suggested that use domain analysis, software evolution, and design patterns. However, identifying the required flexibility for the family of applications and designing mechanisms that provide this flexibility is the problem. Furthermore, the evolution of the framework must be considered, especially as all frameworks seem to mature from initial versions through to a stable platform.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SSR'01, May 18-20, 2001, Toronto, Ontario, Canada.  
Copyright 2001 ACM 1-58113-358-8/01/0005 ...\$5.00.

Refactoring of source code is one approach suggested for the development and evolution of frameworks. Refactoring as a concept or method is gaining in recognition through the publicity of extreme programming.

We wish to show how a framework can be described using a set of models, not just source code, and how refactoring as a concept can be broadened to apply to each of these models. Our view of refactoring includes the underlying motivation for the restructuring performed during the refactoring. This motivation could be expressed as an issue, rationale, or force (or sets of these). In overview, *cascaded refactoring* is the process of sequentially refactoring each model, one after the other, where the restructuring of the previous model provides the motivation for the refactoring of a model.

We explain the general context of our methodology for framework evolution, but the real focus of this paper is on cascaded refactoring as a concept. Evolution is viewed as refactoring followed by extension, but we do not address the extension step here.

The layout of the paper is to present the background on frameworks, framework evolution, and refactoring; then to briefly explain our methodology and models. The main part of the paper discusses the concept of cascaded refactoring and how to refactor the various models. We finish by discussing related work and our conclusions.

## 2. BACKGROUND

A framework exists to support the development of a family of applications. Reuse involves an application developer, or team of application developers, customizing the framework to construct one concrete application. Typically a framework is developed by expert designers who have a deep knowledge of the application domain and long experience of software design.

A framework [12] is a collection of abstract classes that provides an infrastructure common to a family of applications. The design of the framework fixes certain roles and responsibilities amongst the classes, as well as standard protocols for their collaboration. The variability within the family of applications is factored into so-called "hotspots", and the framework provides simple mechanisms to customize each hotspot. Customizing is typically done by subclassing an existing class of the framework and overriding a small number of methods. Sometimes, however, the framework insists that the customization preserves a protocol of collaboration between several subclasses, so customization requires the parallel development of these subclasses and certain of their methods.

A framework evolves through several stages of maturity [23] as the developers increase their understanding of the domain and the required customizations:

- White-box framework: the application developer creates subclasses in order to customize the framework and needs to look at the code of the abstract classes being subclassed.
- Component library: many concrete subclasses are available for selection by the application developer, so much fewer new subclasses are created during customization.
- Pluggable object: extensive use is made of delegation and there are concrete subclasses available to serve as the targets of delegation, so the application developer can customize by parameterizing subclasses.
- Fine-grained object: the role or functionality of classes is decomposed into smaller classes, allowing more mix-and-matching of pluggable objects because of their finer granularity.
- Black-box framework: the application developer does not need to look at the code internals in order to customize, instead existing fine-grained components are selected and composed.
- Visual builder: the choice of components and their composition is done using drag-and-drop in a graphical interface.

Evolution is naturally viewed as refactoring followed by extension in those bottom-up methodologies for framework development that are centered on refactoring. The new structure of the system and the refactorings to effect the restructuring are chosen with the required extension in mind. However, there is little literature on this connection between refactoring and extension. We also say very little in this paper. A useful reference on extension of object-oriented systems is Mätzel and Bischofberger [16].

Refactoring [27] is a behavior-preserving program transformation that automatically updates an application's design and underlying source code. Primitive refactorings perform simple edits such as adding new classes, creating instance variables, and moving instance variables up the class hierarchy. Compositions of refactorings can create abstract classes, capture aggregation and components [18], and even install design patterns [27].

Opdyke [18] introduced the term refactoring and defined a set of behavior preserving transformations for object oriented applications based on the work by Banerjee and Kim [1], the design principles of Johnson and Foote [12], and the design history of the UIUC Choices software system [15]. Roberts [22] developed the Smalltalk Refactory Browser which implements many of these refactorings. Tokuda and Batory [28] proposed additional refactoring to support design patterns as target states for software restructuring efforts. Extreme programming, and Fowler's book [9] have increased the visibility of refactoring of source code within the community.

Tokuda [29] divides the kinds of architectural changes offered by source code refactoring of object oriented systems into:

- schema transformations, which change class structure;

- introduction of design patterns as micro-architectures; and
- hotspot generalization.

John McGregor and his colleagues [17] have developed *use case assortment* for the requirements analysis phase of framework development. The use case model is factored by introducing abstract actors and abstract use cases, and re-arranging responsibilities. This allows the use case model to reflect the commonality/variability analysis. This work struck us as the germ of refactoring of use case models, similar to the refactoring of a class hierarchy, and inspired our work.

### 3. OUR METHODOLOGY AND MODELS

Our methodology for the development and evolution of application frameworks is a hybrid approach combining the modeling aspects of top-down domain engineering approaches, and the iterative, refactoring approaches of the bottom-up object-oriented community. The basic philosophies are

*Framework development is framework evolution.  
Framework evolution = framework refactoring +  
framework extension.*

The set of models that we use are:

- a feature model organizing common and variable features;
- a use case model of requirements including variation points;
- an architectural design with hotspots;
- a design showing collaborations and the overlapping of roles from design patterns onto the hotspots; and
- the source code, with classes, hooks, and templates.

The methodology weaves together steps for partial domain engineering to better understand the domain and how to evolve the current working set of partial models, and steps of system refactoring and extension.

The methodology stresses *traceability* between the models. In this respect it follows FeatureRSEB [10] and FORM [13]. In the context of a framework or product line, the methodology must also stress the commonality-variability distinction. We define an *alignment* [4] of models to be a traceability mapping that is consistent with the mapping of common and variable features.

While the working set of partial models is incomplete, and hence some mappings for traceability or alignment will be partial maps, we still desire a *consistent, coherent, aligned set of models and maps*.

The research into the methodology is still in its infancy, and the details of the methodology will change and be fleshed out as our experience with its use for the development and evolution of our KNOW-IT-ALL framework for database management systems. Still, it is clear to us that there are two dimensions of evolution that the methodology must eventually address: the first dimension is the usual one of maturity of the applications within the product line; the second dimension is the maturity level of the framework as it evolves from white-box through plug-and-play and black-box to application generator. It is not clear whether the later evolutionary

jumps, particularly the last jump to application generator, will necessitate different solutions from our proposed refactoring and extension. Perhaps, at each stage of maturity, there are different issues to be considered when evolving the framework, but our experience with frameworks over their entire life-cycle is rather limited, and it is too early to draw conclusions.

## 4. CASCADED REFACTORING

Our aim is to extend the notion of refactoring that has been applied to source code, and to design in the form of class diagrams, to other models of software systems. The contributions we make in this section are

- refactoring of feature models;
- refactoring of use case models;
- preliminary work on refactoring of architectures; and
- clarify the “behaviour” that is preserved by these refactorings.

Another aim is to relate the set of refactorings across the set of models, as an initial step in describing a methodology for refactoring. The main aim is to guide the choice of refactorings. The contributions we make in this section are

- to view refactoring as an issue-driven activity;
- to document the rationale of an application of a refactoring as a triple: intent of restructuring, choice of refactoring(s), and impact of the restructuring; and
- the notion of cascaded refactoring, where the restructuring of one model determines constraints on the restructuring of other models (via the traceability and alignment maps).

The basic questions we seek to answer are the following.

*What notion of behaviour is appropriate for a model?*

Refactoring of source code is a behaviour-preserving transformation, where functionality is seen as primary and to be preserved while other quality attributes such as performance take a secondary role. Should we use functionality as the “behaviour” to be preserved when we restructure feature models, use case models, and architectures? So, what notions of “structure” are important for a particular model, and what is “re-structured”, and what is preserved?

*How does one record the intent of a refactoring?*

If refactorings preserve behaviour, then there must be some other quality attribute that is affected by the restructuring of the model. Often, it is the attribute of “modifiability”, since refactorings are preliminary steps to extensions. But surely, there must be other reasons as well, particularly for architectural refactorings.

*How does one reason that the chosen set of refactorings achieves the desired intent?*

A design rationale records the assumptions, arguments, and decisions behind a particular design. The rationale allows reviewers and maintainers to follow the reasoning used. What rationale should we record to allow similar reasoning about the choice of re-design that is inherent in a choice of refactorings?

*How does one refactor a set of models and preserve their internal consistency, traceability, and alignment?*

Our work uses a set of models to describe a framework, with the intent that the maps should support the process of evolution. So, if we wish to adopt refactoring as a step in the evolutionary process, we need to preserve the coordination between the set of models.

In the following subsections that treat the individual models, we will discuss what quality attribute should be preserved, what structure is applicable to the models, and what re-structuring is feasible. We offer an initial list of refactorings. No doubt this list will grow as we learn more about the evolution of frameworks.

### 4.1 Refactoring a Feature Model

A *feature* is any aspect of a software system that is used to characterize the system to the users, customers, or developers. A *feature model* is a collection of all the features and their relationships. For a product line or framework, a feature model classifies each feature as *mandatory*, *optional*, or *alternative* in order to express the commonality and variability across the applications in the product line. An application is specified by its *feature set*, which is the set of all features provided by the application. A feature set is a subset of the set of all features, and the feature set satisfies the constraints amongst features imposed by the model.

Feature models contain features related to functionality, but it also contains a much broader range of features. Hence, it is not sensible to restrict the notion of “behaviour” to just features of functionality. A feature model defines a collection of valid feature sets; a *refactoring is required to preserve the collection of valid feature sets*.

There are several views of structure for a feature model. The basic concept is a finite set  $F$  of the features. The usual relationship of interest is the *subfeature* relationship that defines a hierarchy of features. There is also a *dependency* relationship between features, and the actual dependency may be specified using a constraint. One can consider the subfeature relationship as a special kind of dependency.

Structure is also provided by two classifications. The usual classification into mandatory, optional, and alternative can be considered as a map *variability* from  $F$  to the set { *mandatory, optional, alternative* }. The FORM methodology adds a classification of features into capability features, domain technology features, operational environment features, and implementation features. This can be considered as a map *kind* from  $F$  to the set { *capability, technology, environment, implementation* }.

The first set of refactorings modify the variability of a feature:

**change\_mandatory\_to\_optional** takes a mandatory feature  $f$  and makes it optional. This does not invalidate any feature set that was valid under the previous feature model. Yet it adds flexibility to the product line.

**change\_optional\_to\_alternative** takes an optional feature  $f$  and enumerates the values  $f_1, f_2, \dots, f_n$  that it can take. This does not invalidate any feature set that was valid under the previous feature model. Yet it adds flexibility to the product line.

**change\_optional\_to\_mandatory** takes an optional feature  $f$  and makes it mandatory. This requires an enabling condition that every existing application in the product line has feature  $f$ .

**change\_alternative\_to\_optional** takes a set of alternative features  $f_1, f_2, \dots, f_n$  and makes an optional feature  $f$ ,

thus allowing a broader range of alternatives for this feature.

**add\_alternative** takes a set of alternative features  $f_1, f_2, \dots, f_n$  and adds another feature  $f_{n+1}$  to the list of alternatives.

The second set of refactorings restructure the feature-subfeature hierarchy.

**promote\_feature** takes a feature  $sf$  which is a subfeature of  $f$  and promotes it in the hierarchy to be a sibling of  $f$ .

## 4.2 Refactoring a Use Case Model

Use cases capture the functionality of a system, called the *target system*, as it is meant to behave in a given environment called the *host system*. A use case describes how a group of external entities, called *actors*, make use of the system under consideration. The use they make is modeled by the passing of signals or information between the actors and the system.

A *use case* is a description of a cohesive set of dialogues that the primary actor initiates with a system. The dialogues are *cohesive* in the sense that they are related to the same task, or form part of the same transaction. Cohesiveness is often determined by having a *goal* in common for the tasks, or by having a common *responsibility*.

The functionality of the system is defined as the set of dialogues. It is the only appropriate thing to preserve when restructuring. *A refactoring of a use case model preserves the set of dialogues of the target system.*

The high-level view of use cases provides a *generalization* and *inclusion* relationship between use cases. Abstract use cases may occur in the model to express commonality amongst use cases, even if the abstract use case has no concrete scenario; its (leaf) children in the generalization hierarchy will have concrete scenarios. The inclusion relationship connects a task with a subtask represented as a use case.

Extension points and variation points in a use case model allow the variability in functionality to be depicted.

A use case encompasses a collection of scenarios, since there may be several ways in which an actor can (successfully or unsuccessfully) attempt a task. Potts [19] defines a *scenario* as “a description of one or more end-to-end transactions involving the required system and its environment”. Scenarios may be classified as follows. The *main scenario* describes the usual way in which the task is successfully performed. Typically, in the main scenario, the simplest sequence of interactions is described, and it is assumed that all steps execute successfully. A *variant scenario* describes another way of using the system where it is assumed that all steps execute successfully. An *exceptional scenario* describes a scenario where exceptional or error conditions may arise. It may be possible to recover from the exceptions and therefore successfully complete the task — this is called a *recovery scenario* — or it may not be possible to recover — this is called a *failure scenario*.

A scenario may be described in several ways, from a simple narrative text description, to numbered steps indicating the subject-action-object triples, to basic Message Sequence Charts (MSCs) [21] or UML sequence and collaboration diagrams.

A use case may also be described in terms of *episodes*. Each episode represents a subtask, or the parts of the dialogues in the scenario that perform the subtask. A high-level Message Sequence Charts (MSCs) [21] can be used to depict

a use case and its episodes utilising operators for sequence, alternation, iteration, parallel execution, and exceptions.

Use cases may be classified into different kinds, such as business use case, requirements use case, analysis use case, system use case, and design use case. These reflect the perspective of the writer, as well as the level of detail and the kind of target system.

The refactorings of McGregor that inspired this work are:

**create\_abstract\_actor** identifies two actors  $a_1$  and  $a_2$  with a common super-actor  $a$  as their parent.

**create\_abstract\_usecase** identifies two use cases  $u_1$  and  $u_2$  as specializations of a common super-use case  $u$ .

**merge\_actors** identifies two actors  $a_1$  and  $a_2$  as a common actor  $a$ .

**merge\_behaviours** identifies two use cases  $u_1$  and  $u_2$  as a common use case  $u$ .

The refinements of Catalysis are the inverse refactorings of the above two refactorings.

**split\_behaviour** distributes a set of scenarios for a use case  $u$  across two new use cases  $u_1$  and  $u_2$ .

**split\_actor** identifies the special cases  $a_1$  and  $a_2$  of an actor  $a$ .

The next set of refactorings re-distribute behaviour in the form of scenarios or episodes from one use case to another. These are similar to the refactorings of a class hierarchy that move methods.

**make\_episode\_usecase** takes a usecase  $u$  with an episode  $e$  and creates a new usecase  $u_1$  with behaviour precisely that of the episode  $e$ . A relationship link  $u$  includes  $u_1$  is added.

**make\_scenario\_usecase** takes a usecase  $u$  with a scenario  $s$  and creates a new usecase  $u_1$  with behaviour precisely that of the scenario  $s$ . A relationship link  $u$  includes  $u_1$  is added.

The generalization hierarchy can be restructured similarly to a class hierarchy.

## 4.3 Refactoring an Architectural Model

Software architectures provide an abstract description of the organisational and structural decisions that are evident in a software system. The development of an architecture requires the decomposition of system into subsystems, the distribution of control and responsibility, and the development of the components and their connections or means of communication. General principles of information hiding, such as the use of modules, layers, and abstract machines (API's) help manage the complexity of the task.

So many quality attributes are relevant to architectures, it is not clear which properties should be preserved by transformations. Probably, one wants a set of quality-preserving refactorings for each quality attribute. We will take the default quality of functionality. *A refactoring of an architecture preserves the functionality of the system.*

An architectural style [25] may be described in terms of

- a *vocabulary* of the basic design elements (components and connector types),
- a set of *configuration rules* which constrain how components and connectors may be configured,
- a *semantic interpretation* which defines when suitably configured designs have a well-defined meaning as an architecture, and

- *analyses* that may be performed on well-defined designs.

From the perspective of UML and RUP [14], the description comprises four diagrams plus the use cases/scenarios. The diagrams cover the design view, the process view, the implementation view, and the deployment view. The design view presents the subsystems, their interfaces, the dependency between subsystems, and nesting of subsystems.

It is possible to take a use case view of a subsystem, where the subsystem is regarded as the target system with the host system consisting of all other subsystems in the architecture. Then a subsystem's services are described in the use case model for the subsystem, and accessed through its interfaces.

It is also possible to take a class view of a subsystem, where one identifies a subsystem with a facade class. One identifies the subsystem interfaces with the class methods.

Our work on refactoring of architectures is still preliminary. We focus on the traditional subsystems and interfaces view of an architecture.

The first set of refactorings looks at the interfaces of a subsystem and re-distribute their methods.

**split\_interface** takes an interface  $I$  of a subsystem  $S$  and redistributes the methods of  $I$  across two new interfaces  $I_1$  and  $I_2$  of the subsystem  $S$ .

**merge\_interfaces** is the inverse transformation.

The second set of refactorings re-distribute the services provided by a subsystem. Usually these are accompanied by a re-distribution of interfaces. **structure**

**move\_service\_to\_sibling** takes a service  $s$  of subsystem  $S_1$  with sibling subsystem  $S_2$  in a hierarchical client-supplier architecture and assigns it to subsystem  $S_2$ .

**delegate\_service\_to\_supplier** takes a service  $s$  of subsystem  $S_1$  with a supplier subsystem  $S_2$  and assigns the service to  $S_2$ .

**promote\_service\_from\_internal** takes as service  $s$  provided by a nested subsystem  $S_2$  of a subsystem  $S_1$  and makes it a service of  $S_1$ .

To this list of refactorings one can add those that introduce design patterns [27].

## 4.4 Documenting Refactoring

It is important to capture the rationale behind the restructuring of the system. The re-structuring is effected through the refactoring of the models, so the purpose of each refactoring and the reasons for its choice are part of the rationale.

The overall rationale is a collection of decisions. Each decision records the arguments behind a choice of refactoring. We suggest that the decision record the following information:

- the *intent* of this step of restructuring, perhaps in the context of the overall intent;
- the *choice* of refactoring, or refactorings for a high-level refactoring that is composed of a series of lower-level refactorings; and
- the *arguments* for this choice, with a possible discussion of trade-off analysis for other candidate refactorings; and
- the *impact* or consequences of this step of restructuring.

Of course, this rationale should relate to the assumptions and priorities documented in the original design rationale, and the planned extension.

## 4.5 Cascading across the Models

This section elaborates on *cascaded refactoring*. For the set of models, we view the refactoring of the whole set as a cascade through refactoring of each model where the restructuring of one model determines constraints on the restructuring of other models (via the traceability and alignment maps).

Our models are

- $M_f$ , the feature model;
- $M_u$ , the use case model;
- $M_a$ , the architectural model;
- $M_d$ , the design or class hierarchy; and
- $M_s$ , the source code.

We also have a test plan and test cases as part of the documentation, but we will ignore them for the present.

Our alignment maps are

- $t_{fu}$ , the trace map from the capability features to the use case model;
- $t_{fa}$ , the trace map from the operational environment features to the architectural model;
- $t_{fd}$ , the trace map from the domain technology features to the design;
- $t_{fi}$ , the trace map from the implementation features to the source code;
- $t_{ua}$ , the trace map from the use cases to the architecture;
- $t_{ud}$ , the trace map from the use cases to the design;
- $t_{ad}$ , the trace map from the architecture to the design; and
- $t_{di}$ , the trace map from the design to the source code.

The process of *cascaded refactoring* is a series of refactorings of the models  $M_1$  to  $M_5$ . The impact of the refactorings for a model  $M_i$  is translated via the trace maps that have domain  $M_i$  to determine constraints on the refactorings of models  $M_j$  with  $j > i$ .

## 5. RELATED WORK

There are several categories of methodologies for the development of object-oriented frameworks [8].

Bottom-up methodologies follow an incremental spiral generalization from a small number of applications into a framework [23]. Refactoring of source code is a key step.

- 1: Build first application
- 2: Iterate
  - 2.1 Change impact analysis
  - 2.2 Refactor framework
  - 2.3 Build n-th application

Top-down methodologies borrow heavily from domain analysis, in particular from FODA (Feature-Oriented Domain Analysis), now evolved into FORM [13], which introduced feature models to organize commonality and variability within the domain.

- 1: Domain analysis
- 2: Develop domain specific software architecture (DSSA)
- 3: Implement DSSA
- 4: Populate DSSA as applications required

Organizational Domain Modeling (ODM) [26] is a detailed domain analysis process with a set of workproducts and dossiers. Lucent has developed the FAST (Family-Oriented Abstraction, Specification, and Translation) methodology for product lines [30]. FAST promotes very small product lines, which are well-understood, so development is a one-increment activity. The SEI has a program for product lines that is producing a guide to best practice, the Framework for Product Line Practice [5].

Of course, hybrid top-down and bottom-up approaches are possible, where one considers the development of a few applications at once and performs a partial domain analysis.

- 1: Interleave
  - 1.1 Partial domain analysis
  - 1.2 Change impact analysis
  - 1.3 Refactoring
  - 1.4 Build n-th application

Hotspot-driven methodologies identify units of variability, called *hotspots*, and design their system to incorporate flexibility for each hotspot, usually by adopting an appropriate design pattern. The method of *generalization of hotspots* [20, 24] traces the variability in the requirements (the so-called hotspots) to abstract classes in the design: one abstract class per hotspot. There is a variation of this by Demeyer et al [6] that introduces a separate abstract class for each dimension of variability of the hotspot.

- 1: Identify hotspots
- 2: Classify variability required for each hotspot by selecting a meta-pattern
- 3: Associate subsystem to each hotspot
  - 3.1 select design pattern for subsystem
  - 3.2 subsystem generalizes facade

Use-case-driven methodologies, such as Catalysis [7], RSEB [11], and Feature RSEB [10], identify variability in functionality within the use case model. They may then proceed as in hotspot-driven or top-down methodologies. These methods are quite rigorous, they use UML, and stress traceability. Feature RSEB [10] extends RSEB to include a feature model, as in the FODA domain analysis method.

The Design Maintenance System (DMS) [2, 3] is a transformation system that is rule-based. It works with a hierarchy of domains, each specified by a syntax, semantics, and mappings to other domains (or to the same domain). DMS can implement source code transformations: DMS has been used for transformation of COBOL programs for the removal of duplicate code and dead code. In general, the DMS transformations do not have to be behaviour-preserving.

It is feasible within DMS to define a domain corresponding to each of our models, to define a set of transformations for the models, and to define the alignment maps between models. Thus, cascaded refactoring could be realized within DMS. However, this has not been done.

## 6. CONCLUSION

Refactoring of source code has long been used for bottom-up development and evolution of object-oriented frameworks. In this paper, we extend the concept of refactoring to other

models, such as the feature model, use case model, and architecture, of an object-oriented framework. Refactoring of these models is a first step of the two steps in framework evolution: refactoring and extension.

In our methodology, a framework is described by a set of models: a feature model organizing common and variable features; a use case model of requirements; an architectural design; a design showing collaborations and the overlapping of roles; and the source code, with classes, hooks, and templates. While the working set of partial models is incomplete, and hence some mappings for traceability or alignment will be partial maps, we still desire a consistent, coherent, aligned set of models and maps.

We introduce the concept of cascaded refactoring, where the overall refactoring of the framework is a set of refactorings of the models, and the constraints on how to refactor a particular model is determined or impacted by the previous refactorings. Hence, the restructuring cascades from one model to the next.

The cascading of impacts of changes follows the alignment and traceability maps between models.

Overall, a refactoring of a framework is a set of model transformations that maps a coherent set of aligned models to another coherent set of aligned models.

Our work is ongoing, particularly in terms of enumerating all the refactorings of the particular models, and in investigating whether there is a need for architectural refactorings that preserve quality attributes other than functionality.

## 7. ACKNOWLEDGMENTS

This work has been supported by the Natural Sciences and Engineering Research Council of Canada, and *Fonds pour la Formation de Chercheurs et l'Aide a la Recherche* of Québec.

## 8. REFERENCES

- [1] J. Banerjee and W. Kim. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference*. ACM, 1987.
- [2] I. D. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, 1992.
- [3] I. D. Baxter. DMS (transformational software maintenance by reuse): A production research system? In *Proceedings of the Fifth Symposium on Software Reusability*, page 163. ACM, 1999.
- [4] G. Butler. Developing frameworks by aligning requirements, design, and code. In *Proceedings of 9th Workshop on Software Reuse (WISR-9)*, 1999.
- [5] P. Clements and L. Northrop. A framework for software product line practice — version 2.0. Technical report, SEI, CMU, 1999.
- [6] S. Demeyer, T. D. Meijler, O. Nierstrasz, and P. Steyaert. Design guidelines for “tailorable” frameworks. *Communications of the ACM*, 40(10):60–64, 1997.
- [7] D. D’Souza and A. Wills. *Objects, Components, and Frameworks with UML — The Catalysis Approach*. Addison-Wesley, 1998.
- [8] M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors. *Building Application Frameworks*:

- Object-Oriented Foundations of Framework Design*. Wiley, 1999.
- [9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [10] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings Fifth International Conference on Software Reuse*, pages 76–85. IEEE Computer Society, 1998.
- [11] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
- [12] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, July 1988.
- [13] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.
- [14] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42–50, November 1995.
- [15] P. Madany, R. Campbell, V. Russo, and D. Leyens. A class hierarchy for building stream-oriented file systems. In *Proceedings of ECOOP'89*, pages 311–328, 1989.
- [16] K.-U. Mätzel and W. Bischofberger. Designing object systems for evolution. *Theory and Practice of Object Systems*, 3(4), 1997.
- [17] G. Miller, J. McGregor, and M. Major. Capturing framework requirements. In M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley, 1999.
- [18] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
- [19] C. Potts, K. Takahashi, and A. Anton. Inquiry-based requirements analysis. *IEEE Software*, pages 21–32, 1994.
- [20] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1995.
- [21] B. Regnell, M. Andersson, and J. Bergstrand. A hierarchical use case model with graphical representation. In *Proceedings of Second International Symposium on Engineering of Computer-Based Systems*, pages 270–277. IEEE Computer Society Press, 1996.
- [22] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.
- [23] D. Roberts and R. Johnson. Patterns for evolving frameworks. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 471–486. Addison-Wesley, 1997.
- [24] H. A. Schmid. Systematic framework design by generalization. *Communications of the ACM*, 40(10):48–51, 1997.
- [25] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [26] STARS. Organization domain modeling (ODM) guidebook — version 2.0. Technical report, Lockheed Martin Tactical Defense Systems, 1996.
- [27] L. Tokuda. Evolving object-oriented architectures with refactorings. In *Proceedings of ASE-99: The 14th Conference on Automated Software Engineering*. IEEE CS Press, October 1999.
- [28] L. Tokuda and D. Batory. Automating software evolution via design pattern transformations. In *3rd International Symposium on Applied Corporate Computing*, October 1995.
- [29] L. Tokuda and D. Batory. Automating three modes of evolution for object-oriented software architectures. In *5th USENIX Conference on Object-Oriented Technologies (COOTS'99)*, May 1999.
- [30] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering*. Addison-Wesley, 1999.