Politechnika
Wrocławska

WYDZIAŁ INFORMATYKI I ZARZĄDZANIA

INSTYTUT INFORMATYKI STOSOWANEJ

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

Praca Magisterska

Refaktoryzacja Modeli UML
- Wsparcie dla Utrzymania Wykonywalnych Modeli

Łukasz Dobrzański

Promotor: Prof. dr hab. inż. Zbigniew Huzar

Recenzent: Dr inż. Bogumiła Hnatkowska

Wrocław 2005

Streszczenie

Jednym z nieuniknionych, negatywnych efektów ewolucji oprogramowania jest erozja projektu. Refaktoryzacja jest techniką, która ma na celu przeciwdziałanie temu zjawisku poprzez sukcesywne polepszanie projektu oprogramowania, nie zmieniając jego obserwowalnego zachowania. Erozja projektu występuje także w kontekście wykonywalnych modeli UML, tzn. modeli, które są wystarczająco szczegółowe, by mogły być kompilowane do wykonywalnych aplikacji.

Celem pracy było zastosowanie refaktoryzacji do obszaru utrzymania wykonywalnych modeli UML, w tym:

1. dokonanie przeglądu literatury dotyczącej:
    a. refaktoryzacji oprogramowania,
    b. refaktoryzacji modeli UML,
    c. wykonywalnych modeli UML;
2. utworzenie wstępnego katalogu transformacji refaktoryzacyjnych wykonywalnych modeli UML na podstawie wybranych refaktoryzacji kodu;
3. sformalizowanie refaktoryzacji z katalogu z wykorzystaniem języka OCL;
4. zaimplementowanie przykładowej refaktoryzacji w wybranym narzędziu umożliwiającym tworzenie wykonywalnych modeli UML.

Wszystkie cele pracy zostały pomyślnie zrealizowane, a do jej głównych rezultatów należy zaliczyć:

1. Przegląd podejść do refaktoryzacji modeli UML [rozdz. 3].
2. Przegląd i porównanie podejść do wykonywalnego UML-a [rozdz. 4 i 5].
3. Identyfikacja *elementów wyzwalających* (ang. *trigger-element*) dla wszystkich refaktoryzacji kodu z katalogu Fowler-a [rozdz. 2].
4. Mapowanie pomiędzy elementami wyzwalającymi refaktoryzacje kodu a ich odpowiednikami w postaci metaklas z metamodelu UML-a 2.0 oraz z modelu obiektowego narzędzia Telelogic/TAU [rozdz. 6].
5. Identyfikacja i charakterystyka *obszarów refaktoryzacyjnych* (ang. *refactoring area*) w wykonywalnych modelach TAU [rozdz. 6].
6. Zaproponowanie szablonu specyfikacyjnego dla refaktoryzacji [rozdz. 6].
7. Specyfikacja wg szablonu dwunastu przykładowych refaktoryzacji wyzwalanych na elementach z obszaru ESPC (Zewnętrzna Struktura Pasywnych Klas) oraz ESAC (Zewnętrzna Struktura Aktywnych Klas) [rozdz. 7].
8. Implementacja przykładowej transformacji – Usuń Pośrednika (ang. *Remove Middle Man*) – w postaci wtyczki do narzędzia Telelogic/TAU [rozdz. 8].

Spośród tematów potencjalnych dalszych badań [rozdz. 9] należy wymienić następujące:

1. Identyfikacja oraz specyfikacja refaktoryzacji i sytuacji kwalifikujących specyficznych dla wykonywalnych modeli UML.
2. Projekt i implementacja profesjonalnej, przemysłowej wtyczki refaktoryzacyjnej do narzędzia Telelogic/TAU.
3. Generowanie implementacji refaktoryzacji z ich formalnych specyfikacji.
4. Refaktoryzacja wykonywalnych modeli w narzędziu w pełni zgodnym z UML-em 2.0.
5. Specyfikacja refaktoryzacji z użyciem Semantyki Akcji (ang. *Action Semantics*)
6. Systematyczne podejście do odkrywania warunków wstępnych refaktoryzacji.
7. Eksperyment oceniający:
    a. Wpływ automatyzacji refaktoryzacji wykonywalnych modeli na produktywność wytwórców oprogramowania;
    b. Wpływ refaktoryzacji na utrzymywalność (ang. *maintainability*) wykonywalnych modeli UML.

Słowa kluczowe: utrzymanie oprogramowania, wykonywalny UML, refaktoryzacja modeli, transformacja modeli

Master Thesis
Software Engineering
Thesis no: MSE-2005:12
July 2005

UML Model Refactoring

- Support for Maintenance of Executable UML Models

Łukasz Dobrzański

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information

Author:
Łukasz Dobrzański
E-mail: lukasdob@poczta.onet.pl

External advisor:
Prof. Zbigniew Huzar
Institute of Applied Informatics
Faculty of Computer Science and Management
Wrocław University of Technology

University advisor:
Dr Ludwik Kuźniarz
Department of Systems and Software Engineering

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

One of the inevitable negative effects of software evolution is design erosion. Refactoring is a technique that aims at counteracting this phenomenon by successively improving the design of software without changing its observable behaviour. Design erosion occurs also in the context of executable UML models, i.e. models that are detailed enough to be automatically compiled to executable applications. This thesis presents results of a study on applying refactoring to the area of maintenance of executable UML models. It contains an overview of recent approaches to UML model refactoring and to executable modelling, followed by identification of refactoring areas in models built in Telelogic TAU, a state-of-the art UML CASE tool. It proposes a systematic approach to specification of both executable UML model refactorings as well as associated bad smells in models. Additionally, it shows how refactorings can be implemented in Telelogic TAU.

Keywords: software maintenance, executable UML, model refactoring, model transformation

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This is an introductory chapter of the thesis. It is structured as follows: Section 1.1 situates refactoring against a background of software maintenance; Section 1.2 presents both the aim of the thesis as well as objectives and posed research questions. Finally, Section 1.3 outlines the remainder of the thesis.

1.1 Background and Motivation

The background of the thesis is *software maintenance* and a related phenomenon of *design erosion*.

1.1.1 Software Maintenance and Evolution

Software maintenance is one of the key issues in the overall software construction and management. Chapin *et al.* [2001] define *software maintenance* as "the deliberate application of activities and processes (…) to existing software that modify either the way the software directs hardware of the system, or the way the system (…) contributes to the business of the system's stakeholders." In the modern approach, software maintenance encompasses activities and processes involving existing software not only after its delivery but also during its development. Worth mentioning is the fact that nowadays more than 80% of total software life-cycle costs is devoted to its maintenance [Pigoski 1997].

Swanson [1976] distinguishes and describes three kinds of software maintenance:

1. *Corrective maintenance* – performed in response to processing, performance and implementation failures;
2. *Adaptive maintenance* – performed in response to changes in data and processing environments;
3. *Perfective maintenance* – performed to eliminate processing inefficiencies, enhance performance, or improve maintainability.

Chapin *et al.* [2001] argue that in practice it is often difficult to classify unambiguously a software maintainer's work to one of the Swanson's categories. These difficulties result from the fact that this taxonomy is (1) too coarse-grained and (2) based on the intensions. Therefore, proposed is a clarifying redefinition of the types of software maintenance and a new semi-hierarchical classification that bases on "objective evidence of maintainer's activities [ibid.]." Their categorisation groups twelve types of software maintenance into four clusters, which are gathered in a decision tree shown in **Figure 1.1**.

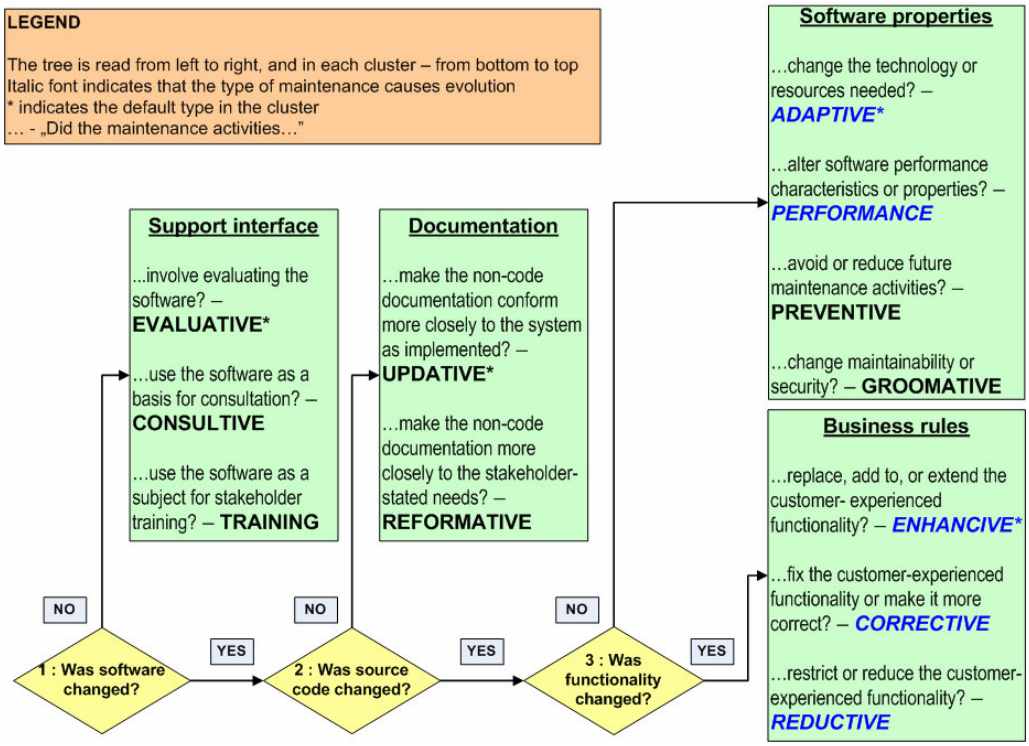

Figure 1.1. Software maintenance classification [Chapin *et al.* 2001]

According to Bennett & Rajlich [2000], currently there is no one commonly accepted definition of *software evolution*, and in a wide sense, it is often used as a synonym of software maintenance. However, Chapin *et al.* [2001] distinguish between software maintenance and software evolution. In their opinion, the latter one occurs when *enhancive*, *corrective*, *reductive*, *adaptive* or *performance* maintenance is carried out. In other words, software evolution happens when business rules, or software properties that are sensible for customer, are changed.

1.1.2 Design Erosion

As indicated by Van Gurp & Bosch [2002], despite many years of research and many suggested approaches, it is inevitable that a software system finally erodes under the pressure of ever-changing requirements. This negative effect of software evolution is known in the literature as *software aging* [Parnas 1994], and one of its dimensions is *design erosion*.

During evolution, almost each change of requirements imposed on a software system enforces the introduction of small adaptations to its design. These adjustments are taken in the context of (1) all previous changes, and (2) predictions about possible future changes that may need to be made. It is obvious that some of these predictions can be wrong. As a consequence, the system may evolve in a direction where it is hard to make the necessary adjustments. Two trends that were observed by Van Gurp & Bosch [2002] are that (1) fixing design erosion is expensive, and (2) eroded software may become an obstacle to further development.

What can be done to stop or at least delay software erosion? As a solution to the problem of software aging, Parnas [1994] suggests (1) designing for change, (2) paying more attention to documentation and design reviews, and (3) not implementing before a proper design is available. Svahnberg [2003] suggests a complementary approach, which relies on establishing a software architecture that is flexible (variable)

in the right places. To avoid taking bad architectural and design decisions, one can furthermore use architectural styles, architectural patterns, and design patterns.

Another approach is to pursue separation of concerns, what causes that effects of changes can be isolated. For instance, by separating the concern synchronisation from the rest of the system, changes in the synchronisation code will not affect the rest of the system. Some examples of approaches towards achievement of separation of concerns are Aspect Oriented Programming, Subject Oriented Programming, and Multi-Dimensional Separation of Concerns. Van Gurp *et al.* [2002] propose an architecture-level design notation that is aimed at modelling concerns on an architecture level while preserving information about the decisions taken during the architecture design.

1.1.3 Refactoring

Another approach to software evolution is represented by agile software development methodologies such as eXtreme Programming (XP) [Beck 2000]. These methodologies advocate that one should develop only for current requirements, as it is impossible to predict what may be required of a software system a few years or more from now [Beck & Fowler 2001]. In XP, the process of implementation is interleaved with the process of *refactoring*, which main goal is to safely and stepwisely improve the design of an existing application without altering its externally observable behaviour, what enables controlled transformation between any two designs.

Lehman [1980] argues that proper maintenance work can avoid or at least postpone the decay. In the context of Swanson's classification [1976], refactoring is an activity that supports a subset of perfective maintenance that aims at software maintainability improvement. In the context of the categorisation provided by Chapin *et al*. [2001], refactoring is an activity that directly supports the types of maintenance present in the software properties cluster. It is particularly suitable for *groomative* maintenance that involves among others "replacing components or algorithms with more elegant ones, (…) changing data naming conventions, altering code readability or understandability [ibid.]." Refactoring can be also useful in *preventive* maintenance activities, which example is "participation in software reuse [ibid.]." *Adaptive* maintenance encompasses activities like "reallocating functions among components or subsystems, (…) and changing design and implementation practices [ibid.]", what also can be achieved by the use of refactoring. The only type of maintenance from software properties group that is not directly supported by refactoring is performance, although some of refactoring transformations (e.g. Inline Method) can be used in this category as well.

Refactoring supports neither the three kinds of software maintenance from support interface cluster, nor both types from documentation one, since they all occur when source code is not changed. As refactoring preserves system's externally observable behaviour, it is also inadequate in the case of the three types of software maintenance from business rules cluster. However, in the work on problems and causes of design erosion, Van Gurp & Bosch [2002] identify two extremely different strategies for incorporating change requests into a software system, namely *minimal effort strategy* and *optimal design strategy*. The former approach relies on incorporating changes in the next iteration of the development while preserving as much of the old system as possible, and in the latter one, all the necessary changes to the software artefacts are made in order to get an optimal system for the new set of requirements. Refactoring very well fits to the optimal design strategy. Therefore, refactoring software before

performing its enhancive maintenance can significantly reduce effort needed to enhance the customer-experienced functionality.

1.1.4 UML Model Refactoring

Besides many theoretical advantages of refactoring software in the form of UML models over programming language code, which result mainly from the possibility to specify systems on a higher level of abstraction and to present them visually, this approach has at least one key drawback. Namely, the majority[1] of research papers on UML model refactoring concern models built with the use of UML 1.4 or one of its earlier versions, what implies that no formally defined behaviour of these models can be specified. This lack of formal specification of model's behaviour is contradictory with a requirement imposed by a definition of refactoring [Fowler *et al.* 1999] stating that the transformation does not change system's observable behaviour, what – in this case – obviously cannot be proven[2]. Moreover, this approach carries another large problem, namely vertical consistency between refactored models and code.

One of the solutions of these issues is to use a UML profile that is extended to enable modelling information required for source-consistent and behaviour-preserving refactoring. An example of such an extension is *GrammyUML* provided by Van Gorp *et al.* [2003], which allows modelling certain constructs that may occur in method bodies. However, this approach does not solve another problem of prior studies on UML model refactoring that concerns passing of the effects of a transformation to the underlying source code. Moreover, this approach still does not enable to perform on the model level many "fine grained" code refactorings.

An approach free of the issues described in the preceding paragraphs relies on using an executable UML, which is considered to be a major innovation in the field of software development. It is a graphical specification language, which combines a streamlined, computationally complete subset of the UML with executable semantics and timing rules. In contrast to traditional specifications, an executable specification can be run, tested, debugged and measured for quality attributes. However, the main benefit of this approach is the possibility of fully automated translation of executable UML models into source code. Executable models confer independence from the software platform, which makes them portable across multiple development and execution environments.

The phenomenon of design erosion occurs also in the context of executable UML models, which are the primary artefacts in e.g. Agile MDA [Mellor *et al.* 2004] software development methodology and thus have to be maintained throughout the whole lifetime of modelled applications.

1.2 Research Questions and Objectives

Although the concept of refactoring is being researched more and more thoroughly, to the knowledge of the authors there is only one research paper that takes the subject of refactoring of executable UML models [Kazato *et al.* 2004]. The aim of this thesis is to fill up this gap by **applying refactoring to the area of maintenance of executable UML models**.

[1] Generally all except for one – [Kazato *et al.* 2004].
[2] Assuming that the system's behaviour is specified in the underlying code.

The thesis addresses following **research questions**:

1. What is the state-of-the-art in UML model refactoring?
2. What is an executable UML model and how can it be built and executed?
3. Which code refactorings are applicable for executable UML models?
4. How can one specify executable UML model refactorings?
5. How can one automate the processes of applying executable UML model refactorings in a state-of-the-art UML CASE tool?

The **objectives** of the thesis are as follows:

1. Perform literature search and review on:
    a. software refactoring,
    b. UML model refactoring,
    c. executable modelling with UML.
2. Create an initial catalogue of executable UML model refactorings consisting of their informal specifications.
3. Formalize the refactorings from the catalogue with the use of Object Constraint Language (OCL).
4. Implement an exemplary refactoring from the catalogue in a state-of-the-art UML CASE tool.

The **type of the thesis** is *problem solving*, which is defined by Dawson [2000] in the following way: "(…) this can involve the development of a new technique to solve a problem or might involve improving the efficiency of existing approaches. **It might also involve the application of an existing problem solving technique to a new area**." In the context of the thesis, the problem is design erosion, the existing problem solving technique is refactoring, and the new area is maintenance of executable UML models.

1.3 Outline of Thesis

Although Opdyke [1992] adopted the scientific approach to the idea of refactoring over a decade ago, it gained a deserved researchers' attention just in 1999, when Fowler published book entitled *Refactoring: Improving the Design of Existing Code* [Fowler *et al*. 1999]. The concepts of UML model refactoring and Executable UML are even more innovative – first papers concerning UML model refactoring appeared in 2001 [e.g. Sunyé *et al.* 2001; France & Bieman 2001] and first books about Executable UML in 2002 [Starr 2002a; Mellor & Balcer 2002].

The novelty of above-mentioned ideas, which lay the basis for the research, motivates to provide thorough background information on software refactoring (Section 2), UML model refactoring (Section 3), and executable UML (Section 4). The remainder of the thesis is structured as follows: Section 5 provides an overview of executable UML models that can be built and executed in Telelogic TAU; Section 6 presents the results of an initial study on refactoring executable UML models; Section 7 constitutes an initial catalogue of specifications of refactorings of TAU executable models; Section 8 shows how refactorings specified in the previous section can be implemented in TAU. Finally, Section 9 concludes the thesis and points out possible directions of a future work.

2 REFACTORING

The main goal of this chapter is to present a literature survey on software refactoring. It is structured as follows: Section 2.1 provides an overview of definitions of refactoring; Section 2.2 presents classifications of refactoring transformations; Section 2.3 outlines the refactoring process; Section 2.4 presents benefits of refactoring; Sections 2.5 and 2.6 introduce notions of behaviour preservation and bad smells, respectively. Finally, Section 2.7 concludes this chapter with identification of among others *trigger-elements* of refactorings.

It is noteworthy that an extensive literature survey on software refactoring [Mens *et al.* 2002; Mens & Van Deursen 2003; Du Bois *et al.* 2004; Mens & Tourwé 2004] has been performed in the framework of "The Refactoring Project"[3] at University of Antwerp.

2.1 Introduction

The term *refactoring* probably originates from Deutch's quote, who wrote "interface design and functional factoring constitute the key intellectual content of software [Deutch 1989, cited by Roberts 1999]." Therefore, **re**factoring can be understood as a process of internal **re**distribution of functionality provided by a software system. In the context of object-oriented paradigm, this redistribution concerns classes, attributes and operations, which are the carriers of software functionality.

The first definition of refactoring was provided by Opdyke in his PhD thesis [1992] as "program restructuring transformation that supports the design, evolution and reuse of object-oriented application frameworks."

According to Chikofsky & Cross [1990], restructuring is "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics)."

Restructuring and refactoring are very similar terms but according to Fowler [2004], they are not the same - refactoring is a "very specific technique to do the more general activity of restructuring (…) founded on using small behaviour-preserving transformations (themselves called refactorings)." It means that while refactoring, the subject system should not be broken for more than a few minutes at a time.

The term restructuring has also a broader meaning, which does not demand the preservation of system's external behaviour. Its definition, provided by Chikofsky & Cross [1990], recognizes restructuring process as "the application of similar transformations and recasting techniques in reshaping data models, design plans, and requirements structures." This meaning of restructuring is inconsistent with the idea of refactoring, because one cannot refactor something that does not have a well-defined behaviour.

Roberts in his PhD thesis [1999] changed the definition of refactorings to be "program transformations that have particular preconditions that must be satisfied

[3] http://win-www.uia.ac.be/u/lore/refactoringProject

before the transformation can be legally performed." This definition encompasses both behaviour-preserving and non-behaviour-preserving transformations. Additionally, Roberts augmented Opdyke's definition of refactorings by postconditions, which specify how the preconditions, introduced by Opdyke in his thesis [1992], are transformed by the refactorings. More formally, "a refactoring is an ordered triple R = (pre, T, P) where pre is an assertion that must be true on a program for R to be legal, T is the program transformation, and P is a function from assertions to assertions that transforms legal assertions whenever T transforms programs [Roberts 1999]."

Fowler in his book [Fowler *et al.* 1999] provides two definitions of refactoring:

> "**Refactoring** (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour."

> "**Refactor** (verb): to restructure software by applying a series of refactorings without changing its observable behaviour."

The second definition is tautological, because as well software restructuring as refactoring (noun) are, by definition, behaviour preserving.

Fowler's definition of refactoring emphasizes that the purpose of refactoring is to make the software easier to understand (improve its readability) and modify (improve its maintainability). That is why e.g. performance optimization, which usually alters only the internal structure of software and preserves its behaviour (excluding its timing characteristics), cannot be classified as a refactoring.

The second important thing is that refactoring does not change the observable behaviour of the software. The intention of refactoring is neither adding new nor altering existing functionality. In this aspect, refactoring is very similar to restructuring, which "does not normally involve modifications because of new requirements. However, it may lead to better observations of the subject system that suggest changes that would improve aspects of the system [Chikofsky & Cross 1990]."

According to Wake [2003], refactoring is "the art of safely improving the design of existing code. Refactoring provides us with ways to recognize problematic code and gives us recipes for improving it." In Wake's opinion, refactoring:

1. does not include just any changes in a system – refactoring itself does not add new features;
2. is not rewriting from scratch – refactoring makes possible to improve code;
3. is not just any restructuring intended to improve code – refactorings are small and safe transformations;
4. changes the balance point between up-front design and emergent design – refactoring lowers the cost and risk of the emergent approach;
5. can be small or large – hopefully large refactorings are rarely needed.

2.2 Classifications of Refactorings

All refactorings mentioned in this section are understood as *code refactorings* – more precisely as *class-based object-oriented refactorings*, i.e. behaviour-preserving transformations of object-oriented source code written in among others Smalltalk,

C++, or Java. The other niche kinds of refactorings like e.g. imperative, functional or logic ones are not subject of this thesis.

Opdyke [1992] defined in terms of preconditions:

1. twenty three low-level refactorings concerning creating, deleting, changing program entities and moving member variable,
2. three composite (also low-level) refactorings, namely:
    - abstract access to a member variable,
    - convert a code segment to a function,
    - move a class,
3. three high-level refactorings concerning:
    - creating an abstract superclass,
    - subclassing and simplifying conditions,
    - creating aggregations and reusable components.

Roberts's [1999] refactorings, defined in terms of pre- and postconditions, are grouped into three categories: class, method and variable refactorings.

The most extensive catalogue of refactorings is provided by Fowler in his book [Fowler *et al.* 1999]. A description of each refactoring contains following sections: a name, a short summary, a motivation behind the transformation, a mechanics describing how to carry out the refactoring, and examples illustrating its use. All seventy-two refactorings collected by Fowler are grouped into seven categories (see **Table 2.1**). The full list of Fowler's refactorings, containing their problem and solution statements from summary sections, can be found in Appendix A.

Table 2.1. Fowler's refactoring categories [Fowler *et al.* 1999]

Category	Description – Refactorings that…
Composing Methods	…help compose methods to package code properly
Moving Features Between Objects	…let change decisions where to put responsibilities
Organizing Data	…make working with data easier
Simplifying Conditional Expressions	…let simplify tricky conditional logic
Making Method Calls Simpler	…make interfaces of classes more straightforward
Dealing with Generalization	…deal with moving methods around a hierarchy of inheritance
Big Refactorings	…are examples of how to use all other refactorings to refactor to some purpose

An up-to-date list consisting of all refactorings from Fowler's book [Fowler *et al.* 1999] and several dozen other, including all J2EE refactorings described by Alur *et al.* [2001] is maintained by Fowler and available from Internet [1999].

Gamma *et al*. [1995] wrote that design patterns provide targets for refactorings. Fowler has later paraphrased this thought in words: "There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else [Fowler *et al.* 1999]." As indicated by Roberts [1999], design patterns create many opportunities for refactoring that can be used to both introduce new design patterns into existing programs, and to remove the ones which add the flexibility that is not needed.

Five refactorings in Fowler's catalogue belong to the group of so-called *refactorings to patterns*, namely:

- Duplicate Observer Data,
- Replace Type Code with State/Strategy,
- Introduce Null Object,

- Replace Constructor with Factory Method,
- Form Template Method.

Inspired by the work initiated by Fowler, Kerievsky [2004] described twenty-seven high-level refactorings composed of low-level ones that provide insights into implementing design patterns.

2.3 Refactoring Process

The *refactoring process* (cycle) suggested by Wake [2003] consists of three iteratively repeated steps:

1. Identify parts of software that should be refactored.
2. Choose appropriate refactoring(s) to the identified places.
3. Apply the refactoring(s).

Mens & Tourwé [2004] identified three additional distinct activities, so that the complete refactoring process consists of following six steps:

1. Identify parts of software that should be refactored.
2. Choose appropriate refactoring(s) to the identified places.
3. Ensure that selected refactoring(s) will be behaviour preserving.
4. Apply the refactoring(s).
5. Assess the effect of the refactoring on quality characteristics of the software (mainly maintainability) and/or the process (e.g., productivity).
6. Maintain the consistency between the refactored program code and other software artefacts (e.g. requirements specification, analysis models, design models, tests).

Kataoka *et al.* [2002] suggest that to be able to assess whether a particular refactoring it economically justified, the validation of refactoring effect (step 5) should be carried out before application of refactoring(s) (step 4). The approach to the refactoring effect evaluation proposed by Kataoka *et al.* [ibid.] consists of two major steps: (1) selection of appropriate maintainability quantification metrics, e.g. coupling metrics, and (2) measurement and comparison of selected metrics before and after refactoring.

2.4 Motivation for Refactoring

According to Opdyke [1992], in the context of object-oriented systems, refactoring is needed to "refine the design of an already structured program, and make it easier to reuse." He enumerates three cases where refactorings might be applied:

1. extracting a reusable component;
2. improving a consistency among components;
3. supporting the iterative design of an Object-Oriented Application Framework.

Fowler lists four purposes that motivate refactoring in the context of XP [Fowler *et al.* 1999]:

1. improving the design of software – preventing program's design from decay and eliminating duplicate code;
2. making software easier to understand – well factored code better communicates its purpose;

3. helping finding bugs – while clarifying the structure of the program it is easy to spot the bugs;
4. helping programming faster – a good design is conducive to rapid software development.

Developers from XP circles usually use subjective criteria in assessing the quality of software designs, what leads to situations in which a design that is found by some developers to be "good", for others "screams for improvement." It results from the fact that the quality of a software design depends on the quality attributes which one takes into consideration during the assessment. For example, a design that is "good" from the performance point of view can be found "bad" from the maintainability perspective. Reassuming, one can say that the design of software is "good" if it fulfils the quality requirements imposed on the system.

As it is possible to measure or estimate for each piece of software its *external quality attributes*, Mens & Tourwé [2004] argue that refactorings can be classified according to which of these quality attributes they affect. It allows improving the quality of software by applying the relevant refactorings at the right places. Each refactoring has its particular effect (e.g. removal of code redundancy or enhancement of the reusability), which can be estimated by expressing the refactoring in terms of the *internal quality attributes* it affects (such as size, complexity, coupling, and cohesion). According to Mens & Tourwé [2004], some of the techniques that can be used to measure or estimate the impact of a refactoring on quality characteristics are *software metrics*, *empirical measurements*, *controlled experiments*, and *statistical techniques*.

According to Bosch [2000], quality requirements can be categorized as either development or operational quality requirements. *Development quality requirements* are "qualities of the system that are relevant from a software engineering perspective", e.g. maintainability, reusability, flexibility and demonstrability. *Operational quality requirements* are "qualities of the system in operation", e.g. performance, reliability, robustness and fault-tolerance. The goal of refactoring is to improve mainly development quality attributes, however it can positively affect also operational quality ones, including – what astonishing – performance [Demeyer 2002].

2.5 Behaviour Preservation of Refactorings

According to Mens & Tourwé [2004], "the preservation property of a program transformation guarantees that (some aspect of) the program behaviour is preserved by the transformation." Ó Cinnéide [2000] distinguishes three possible approaches to behaviour-preservation, namely (1) a non-formal approach (e.g. [Fowler *et al*. 1999]), (2) a semi-formal approach (e.g. [Roberts 1999]), and (3) a fully formal approach. However, even with the use of the last approach it is impossible to guarantee full behaviour preservation in its generality [Mens & Tourwé 2004].

The idea of behaviour preservation in the context of refactoring was introduced by Opdyke [1992] who defined it intuitively in this way: "if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same." The behaviour preservation of Opdyke's refactorings [ibid.] is argued in terms of the following set of syntactic and semantic program properties:

1. Unique Superclass,
2. Distinct Class Names,
3. Distinct Member Names,
4. Inherited Member Variables Not Redefined,

5. Compatible Signatures in Member Function Redefinition,
6. Type-Safe Assignments,
7. Semantically Equivalent References and Operations.

A compiler can detect violations of the first six properties, but the verification of preservation of the semantic equivalence requires much more effort.

Each primitive refactoring specified in Opdyke's PhD thesis [ibid.] has a set of preconditions, which have to be met in order to ensure that behaviour of refactored program will not be altered. The behaviour-preservation of high-level refactorings results from the fact that they are composed of the proven low-level ones.

Roberts [1999] specified the refactorings' preconditions in first-order predicate logic (FOPL) with the use of the analysis functions that describe relationships between methods, classes, and instance variables. These functions are divided into two categories, namely (1) primitive, and (2) derived, which can be computed from primitive ones.

Mens *et al.* [2005] maintain that for each refactoring one may list a set of behaviour-related and statically verifiable properties that need to be preserved. Three examples of these properties are as follows:

- *Access preservation* - each method implementation accesses at least the same variables after the refactoring as it did before the refactoring;
- *Update preservation* - each method implementation performs at least the same variable updates after the refactoring as it did before the refactoring;
- *Call preservation* - each method implementation still performs at least the same method calls after the refactoring as it did before the refactoring.

As indicated by Van Gorp *et al*. [2003b], a careful selection of pre- and postconditions of a refactoring can guarantee preservation of the above-mentioned properties.

2.6 Bad Smells in Code

According to Beck & Fowler, *bad smells in code* are "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring [Fowler *et al.* 1999]." Fowler's book contains descriptions of twenty-one bad smells in code. For each bad smell, a set of refactorings that can remove it is provided.

According to Wake [2003], code smells are "warning signs about potential problems in code." The word "potential" suggests that smells are not synonymous with problems, but always worthy of an inspection. Wake [ibid.] proposed a standard format for describing smells consisting of: smell's name, symptoms, causes, possible refactorings, payoffs (what will be improved) and contradictions (when not to fix it). Moreover, He refined Fowler's code smells and classified them into two groups, namely (1) Smells Within Classes, and (2) Smells Between Classes, with several subgroups in each one[4].

Another taxonomy of Fowler's bad smells is provided by Mika Mäntylä [2003; 2004], who divided them into five following groups:

1. The Bloaters,
2. The Object-Orientation Abusers,
3. The Change Preventers,

[4] A complete list of Fowler's and Wake's bad smells can be found in Appendix B.

4. The Dispensables,
5. The Couplers.

During the work on refactoring to patterns, Kerievsky [2004] discovered five new code smells that suggest the need for pattern-directed refactorings, namely Conditional Complexity[5], Indecent Exposure, Solution Sprawl, Combinatorial Explosion[6] and Oddball Solution.

Presence of bad smells in object-oriented software hints at its low maintainability, which can be measured with the use of various maintainability quantification metrics. Some of these metrics concern such aspects of software maintainability like *coupling*, *cohesion*, *size and complexity*, or *description* [Kataoka *et al.* 2002]. Therefore, at least theoretically, the enhancement of the software maintainability can be identified with the reduction of bad smells. However, as stated by Beck & Fowler [Fowler *et al.* 1999], "no set of metrics rivals informed human intuition", what implies that not all bad smells can be revealed with the use of metrics. According to Mäntylä [2003], only about half of code bad smells can be effectively measured.

2.7 Towards Better Comprehension of Refactorings

Mechanics sections of Fowler's refactorings are terse and often incomplete. They are rather guidelines how to perform manually refactorings in the most common cases than ready-to-implement precise algorithms. Some of them contain imprecise statements like – in the case of Substitute Algorithm – "get it so that it compiles". Moreover, in many cases their realisation differs depending on the initial state of the code before refactoring. Therefore, an attempt to classify them unambiguously is not a trivial task.

In this section, no refactorings from the Big Refactorings category, i.e. Tease Apart Inheritance, Convert Procedural Design to Objects, Separate Domain from Presentation, and Extract Hierarchy, are taken into consideration due to their ambiguity and complexity.

2.7.1 Trigger-Elements

With the aim of better comprehension of refactorings, their *trigger-elements* are determined, i.e. language elements, on which they can be triggered. From the practical point of view, a trigger-element of a refactoring is the type of a code element on which it can be triggered in an IDE tool. For instance, a trigger-element of Inline Temp, Replace Temp with Query, Split Temporary Variable, and Remove Control Flag is Temporary Variable. These are the refactorings, which one expects to be able to apply after selecting a temporary variable in the body of an operation belonging to a class, while browsing a code.

All trigger-elements have contexts. For example, the trigger-element of Extract Method is a fragment of code, and the context of this element is a body of a method, which in turn belongs to a class.

In many cases, the type of a trigger-element is contained in the name of a refactoring (e.g. the trigger-element of Inline Class is a class). However, it is not the

[5] An equivalent of Wake's Complicated Boolean Expression.
[6] Not an equivalent of Wake's Combinatorial Explosion.

rule since the name can be sometimes misleading (e.g. the trigger element of Remove Setting Method is a field). In several cases, the name of a refactoring contains the name of a role that a trigger-element plays in the transformation and not a trigger-element itself, like in the case of Hide Delegate, where a delegate is a class.

For each refactoring one can also specify the multiplicity of its trigger-elements. This results from the fact that some refactorings require more than one trigger-element, like in the case of Replace Subclass with Fields, which trigger-elements are at least two classes (multiplicity denoted as [2..*]). The default cardinality of trigger-elements is one.

It should be noted that for some refactorings the indication of a trigger-element is ambiguous, what is a consequence of vagueness of descriptions of Fowler's refactorings. For example, the trigger-element of Introduce Foreign Method could be as well either server- or client class as a fragment of code being an invocation of a constructor of a server class in the body of a method of a client class.

Fowler's refactorings are triggered by following elements:

- Association (Field)
- Generalization (Class [2])
- Record
- Class
- Class::Field
- Class::Method
- Class::Method::MethodBody
- Class::Method::MethodBody::CodeFragment
- Class::Method::MethodBody::LiteralNumber
- Class::Method::MethodBody::TemporaryVariable
- Class::Method::Parameter

It is noteworthy that elements like *association* and *generalization* do not exist explicitly in Java, which is the language that is used in Fowler's book to illustrate examples of refactorings. Therefore, a field and two classes can replace them accordingly. The only non-object-oriented element is a *record*, which triggers one refactoring (Replace Record with Data Class). Additionally, the distinction between *method* and *method body* is made, where the former one means method declaration (in terms of UML – an operation), and the latter one stays for method implementation (in terms of UML – a method or operation body). Some examples of refactorings that are triggered by a *method* are Move Method and Add Parameter, whereas e.g. Inline Method and Separate Query from Modifier are rather, in opinion of the authors, triggered by *method body*.

Sometimes a trigger-element can be constrained, like e.g. in the case of Introduce Explaining Variable, which trigger-element is constrained by *expression*, what means that the fragment of code triggering the refactoring has to be an expression that returns one value.

All trigger-elements are classified into two disjoint groups: (1) *structural elements* and (2) *behavioural elements*. The *method body* and its internals, i.e. *code fragment, literal number*, and *temporary variable*, belong to the latter group, and the rest of them are structural ones. Refactorings triggered on structural elements are *structure-triggered (S-T)*, and the ones triggered on behavioural elements – *behaviour-triggered (B-T)*.

It is noteworthy that there is difference between "triggerness" and "driveness" of refactorings, which are *triggered on* code elements and *driven by* the presence of bad smells.

2.7.2 Behavioural and Structural Modifications

Another feature that can be determined for code refactorings is the type of modifications that they introduce. The majority of refactorings change as well structure as behaviour of software[7]. Therefore, especially interesting is determination whether particular refactorings involve modifications of only either structural or behavioural parts of programs.

This way, singled out are refactorings, which modification scope is restricted to the method bodies, in which defined are their trigger-elements. Obviously, the trigger-elements of these *lowest granularity (behavioural)* refactorings are behavioural. These are:

- Inline Temp (3),
- Introduce Explaining Variable (5),
- Split Temporary Variable (6),
- Remove Assignments to Parameters (7) – not in all languages,
- Substitute Algorithm (9),
- Replace Magic Number with Symbolic Constant (26),
- Consolidate Duplicate Conditional Fragments (36),
- Remove Control Flag (37),
- Replace Nested Conditional with Guard Clauses (38),
- Introduce Assertion (41) – depending on the way the assertion is realized.

Another group of refactorings form the ones that do not introduce behavioural modifications, namely – *structural refactorings*. However, their unequivocal identification is very difficult, because this feature depends on the realization of the refactoring. Nevertheless, these seem to be:

- Hide Method (52),
- Pull Up Field (57),
- Pull Up Method (58),
- Push Down Method (60),
- Push Down Field (61),
- Extract Interface (64).

The list of all[8] sixty-eight Fowler's refactorings with their trigger-elements and their modification characteristics can be found in Appendix C.

2.7.3 Dependencies between Refactorings

Basing on Fowler's catalogue, five kinds of dependencies between refactorings have been identified (see **Table 2.2**). For instance, Extract Method is a transformation that can be inversed by Inline method, and which may be enabled by Split Temporary Variable or Replace Temp with Query in the case when local-scope variables are modified by the extracted code.

[7] More precisely – structure of both structural and behavioural descriptions of the system.
[8] All besides the ones from Big Refactorings category.

Table 2.2. Dependencies between refactorings

Dependency kind	Meaning
Inverses	Refactoring A inverses refactoring B if a modification introduced by B can be cancelled by application of A, and vice versa.
Includes	Refactoring A includes refactoring B if mechanics of A includes or may include the use of B.
Is enabled by	Refactoring A is enabled by refactoring B if A has a precondition that can be fulfilled by application of B.
Is usually preceded by	Refactoring A is usually preceded by refactoring B if B is usually performed before application of A.
Is usually followed by	Refactoring A is usually followed by refactoring B if B is usually performed after application of A.

A table with Fowler's refactorings and their inter-dependencies can be found in a table in Appendix D. It is noteworthy that this table is by no means complete, because – except for "Inverses – it encompasses only these relations between refactorings that have been identified (in the case of "Includes" and "Is enabled by") and observed (in the case of "Is usually preceded by" and "Is usually followed by") by Fowler.

3 UML MODEL REFACTORING

The main goal of this chapter is to present a literature survey on refactoring of UML models. It is structured as follows: Section 3.1 argues that code refactorings can be applied to UML models; Section 3.2 situates model refactoring against a background of model transformations; Section 3.3 presents advantages of refactoring software in the form of UML models over programming language code; Section 3.4 contains an overview of recent approaches to UML model refactoring; Section 3.5 describes the problem of vertical software consistency; Section 3.6 concerns source-consistent UML model refactoring. Finally, Section 3.7 concludes this paper with an outline of a study on refactoring executable UML models.

3.1 Introduction

Although Fowler's book [Fowler *et al*. 1999] concerns code refactorings, over 60% of them (44 of 72) are illustrated with the use of UML class diagrams. This observation motivates a question whether code refactorings can be applied to UML models. Zhang *et al.* [2004] states that it is obvious that some code refactorings can also be used to transform class diagrams. According to Boger e*t al.* [2003], for some refactorings, like e.g. Extract Method, it is natural to apply them on the code representation level. Other, like Rename Class or Pull Up Method can be applied on code as well as on the model level, and refactorings like Replace Inheritance with Delegation or Extract Interface are more apparent on the model level.

Table 3.1 presents the amount of Fowler's refactorings illustrated with UML in each category. Boldfaced are the rows that correspond to the categories of refactorings that seem to be most naturally applicable at the UML model level.

Table 3.1. The amount of Fowler's refactorings illustrated with UML

No.	Category	No. of refactorings	No. of refactorings illustrated with UML	Percentage of refactorings illustrated with UML
1.	Composing Methods	9	1	11
2.	**Moving Features Between Objects**	**8**	**7**	**87**
3.	Organizing Data	16	11	68
4.	Simplifying Conditional Expression	8	2	25
5.	Making Method Calls Simpler	15	8	53
6.	**Dealing with Generalization**	**12**	**11**	**92**
7.	**Big Refactorings**	**4**	**4**	**100**

It should be noted that when one writes about refactoring of a particular kind of diagrams (e.g. class diagrams), meant is the refactoring of a part of a UML model, which is usually shown on diagrams of this type. The taxonomy of different kinds of diagrams [OMG 2004] provides only a logical organization for them and it does not preclude the mixing of e.g. structural and behavioural elements on one diagram, which would show a state machine nested inside an internal structure. This results from the fact that the boundaries between different kinds of diagram types are not strictly enforced, i.e. there are no metamodels for UML diagrams.

Since most approaches to UML model refactoring presented in research papers, and consequently in this section, concern design models, we will use terms "model refactoring" and "design refactoring" interchangeably.

3.2 Refactoring as Model Transformation

Model transformation can be defined as a mapping of a set of source model(s) onto a set of target model(s), following a set of transformation rules [Sendall & Kozaczyński 2003; Sendall *et al*. 2004].

Sendall & Kozaczyński [2003] list five kinds of model transformations, which automation would greatly improve the productivity of developers and the quality of the models: *refinement*, *model reverse engineering*, *generation of new views*, *application of design patterns* and *model refactoring*.

According to Massoni [2003], *model refactoring* can be defined as "model transformation that improves specific qualities of the model, such as extensibility, making the perfective evolution task more manageable."

France & Bieman [2001] identify two broad classes of model transformations: *vertical* and *horizontal*. In vertical transformations, source and target models are at different levels of abstraction. Two examples of vertical transformations are refinement and abstraction. A horizontal transformation results in target model that is at the same abstraction level as the source one. Horizontal transformations usually occur for two reasons: (1) to improve specific quality attributes and (2) support analysis of models. Model refactoring is a horizontal model transformation that occurs to improve specific quality attributes of models.

Pollet *et al.* [2002] distinguish three types of model transformations:

1. *creational* – usually model import from an external source (e.g. XMI);
2. *endomorphic* – from UML to UML;
3. *exomorphic* – from UML to other formats (e.g. XMI or source code).

Model refactoring belongs to the second group of endomorphic transformations.

According to Porres [2003], there are two main approaches to describe and implement model transformations: *mapping* and *update*. Mapping transformations "translate each element from a source model into zero, one or more elements in a target model", so that the source model is not altered. In contrast, update transformations modify, i.e. add, delete and update source model's elements in place. Model refactoring is an example of a behaviour-preserving update transformation of a small and chosen by the designer subset of the model.

Zhang *et al.* [2004] provide yet another definition of model refactoring based on the one given by Roberts [1999]: "A Model refactoring is a pair R = (pre; T) where pre is the precondition that the model must satisfy, and T is the model transformation."

Sendall *et al.* [2004] distinguish two broad categories of model transformations: *language translation*, where a model is translated into another one in a different language, and *language rephrasing*, where a source model is altered or a new, changed one is produced. Language translation is further sub-divided into *migration* – target model and source model are at the same level of abstraction, *synthesis* – target model is at a lower level of abstraction, and *reverse engineering* – target model is at a higher level of abstraction. Language rephrasing is sub-divided into *normalisation* – reduction

to a sublanguage, *correction* of errors, *adaptation* to new or modified requirements, and *refactoring* – "a model is restructured, improving the design, so that it becomes easier to understand and maintain while still preserving its externally observable behaviour."

3.3 Motivation for UML Model Refactoring

Astels [2002], who investigates refactoring in the context of agile modelling, gives several reasons motivating refactoring in UML. First, it is easier to comprehend software's structure when looking at an UML class diagram rather than at a source code. Furthermore, behavioural aspects of software can be modelled and shown in behavioural UML diagrams. For example, instead of having to trace the call sequence of a given scenario in a code editor and switch between several files, the complete scenario can be expressed in one sequence or collaboration diagram [Boger *et al.* 2003]. Such code visualization can help in detecting bad smells and design flaws, and in presenting the impact of the refactoring on the software. Moreover, manipulation of code on higher level of granularity (i.e. methods, variables and classes) can make refactoring more efficient.

According to France & Bieman [2001], applying refactorings on an abstract view of the system facilitates meeting design goals and addressing deficiencies uncovered by evaluations, i.e. improving specific quality attributes directly on a model. It also enables relatively cheap exploring of alternative decision paths in system's design.

Astels [2002] provides examples of using UML to detect following common bad smells: Data Class, Large Class, Lazy Class and Middle Man. He also shows that the following exemplary refactorings: Move Method, Move Field, Make Inner Class Freestanding, Replace Inheritance With Delegation and Replace Delegation With Inheritance, can be performed easier and faster in a UML CASE tool than in an IDE.

Sunyé *et al.* [2001] mention that the primary advantage of UML over other modelling languages, in the context of model refactoring, is the syntax, which is precisely defined by a metamodel. Therefore, the metamodel can be used to control the impact of a transformation and provide means for ensuring its behaviour-preservation.

3.4 Recent and Current Research

There are several attempts to perform refactoring on models expressed in UML. The most representative of them are briefed in this section. The survey was performed basing on the descriptions of the attempts contained in research papers. Mainly due to the unavailability of described tools, neither a detailed comparison nor a classification of these approaches could be prepared.

3.4.1 UMLAUT

Sunyé *et al.* [2001] attempted to transpose some of Robert's [1999] refactorings to UML models. In the result of their work, they created an initial set of UML class diagram refactorings, consisting of:

1. Addition of features (attributes and methods) and associations to a class;
2. Removal of features and associations from a class;
3. Insert Generalizable Element – addition of a class to inheritance hierarchy;

4. Remove Generalizable Element – removal of a class from inheritance hierarchy;
5. Move Method (from one class to another);
6. Generalization of elements owned by classes, such as attributes, methods, operations, association ends and statecharts;
7. Specialization – the exact opposite of Generalization.

For each refactoring, a textual description of preconditions that have to be satisfied before performing the transformation is provided.

Besides class refactorings, Sunyé *et al.* [2001] describe six novel statechart refactorings:

1. State
    a. Fold Incoming/Outgoing Actions;
    b. Unfold Entry/Exit Action;
    c. Group Sates;
2. Composite State
    a. Fold Outgoing Transitions;
    b. Unfold Outgoing Transitions;
    c. Move State into Composite.

Behaviour-preservation of each statechart refactoring is expressed with the use of pre- and postconditions specified in OCL at the metamodel level. For the sake of simplicity, no details of how each refactoring accomplishes its intent, is given in the research paper. However, the authors suggest the use of their UML general-purpose transformation framework called UMLAUT (Unified Modeling Language All pUrposes Transformer) [Ho 1999; Ho 2000].

3.4.2 Refactoring Browser for UML

Boger *et al*. [2003] focus on such UML model refactorings that apply to structure information of software that is not evident while browsing its source code. In their research, they restricted themselves to activity diagram and statechart diagam next to class diagram refactorings.

Some refactorings of class and all of activity and statechart diagrams described in their paper were implemented in a refactoring browser for UML as a part of the Gentleware tool *Poseidon for UML*. Class diagrams refactorings have been implemented down to the level of method signatures. Refactorings covering the method bodies were omitted due to missing notation for them in UML 1.3, on which metamodel is based Poseidon's repository.

Boger *et al.* identified and implemented following statechart refactorings:

1. Merge States;
2. Decompose Sequential Composite State;
3. Form Composite State;
4. Sequentialize Concurrent Composite State,

as well as two activity diagrams refactorings, namely: Make Actions Concurrent and its opposition – Sequentialize Concurrent Actions.

Behaviour-preservation of each refactoring is defined in a form of preconditions that are evaluated for currently selected model elements. Each precondition is mapped to appropriate messages, which are presented to the user in the case of its violation. These messages correspond to conflicts that are grouped into warnings, indicating that

the refactoring might cause side effects, while leaving the model in a well-formed state, and errors, indicating that the refactoring will break the consistency of the model.

3.4.3 SMW Toolkit

Porres describes in his technical report [2003] how a UML model refactoring can be implemented as a sequence of transformation rules or guarded actions. Each transformation rule consists of five elements: its name, a documentation string, a sequence of formal parameters, a guard defining when the rule can be applied, and a body, i.e. the implementation of the rule. A rule takes one or more model elements as actual parameters and performs a basic transformation action based on these parameters.

Porres presents an execution algorithm for the transformation rules and describes a mechanism that ensures that the transformed models are well formed. However, he does not discuss the behaviour-preservation property of refactorings.

In the absence of a standardized language for model transformations, Porres implements refactorings using SMW – a scripting language based on the Python programming language. In many respects, SMW is similar to OCL, but it additionally provides a set of operations enabling implementation of model transformations. The idea of extending OCL with action features has been already discussed by Pollet *et al.* [2002]. The main advantage of this approach is the possibility of implementing model transformations in one language along with defining their pre- and post-conditions.

Models can be accessed from SMW scripts via a metamodel-based interface. Each metaclass from the metamodel is represented in SMW as a Python class and each element in a model – as an instance of an appropriate class. The classes representing the metamodel have the names, attributes and associations as defined in the UML 1.4 standard [OMG 2002].

In order to validate the execution algorithms and to evaluate how difficult it is to implement new refactorings in practice, Porres constructed – using the SMW toolkit – an experimental, metamodel-driven refactoring tool, and integrated it with an existing UML editor.

3.4.4 C-SAW and GME

Zhang *et al.* [2004] describe an approach to model refactoring with the use of the Constraint-Specification Aspect Weaver[9] (C-SAW) model transformation engine, a plug-in component for Generic Modeling Environment (GME). GME is a UML-based meta-modelling environment that can be configured and adapted from loaded into it meta-level specifications (called the *modelling paradigm*) that define all the modelling elements and valid relationships between them in a particular domain. The UML/OCL meta-metamodel of GME is based on its own specification instead of Meta-Object Facility (MOF). However, an ongoing project incorporates OMG's MOF into GME.

A prototype model refactoring browser operating with the underlying C-SAW has been developed as a plug-in for GME. It provides automation of generic pre-defined refactorings within the GME metamodel domain. Additionally, it enables the

[9] http://www.gray-area.org/Research/C-SAW

specification of user-defined refactorings of both generic and domain-specific models (e.g. Petri Nets, AQML models, or finite state machines).

A list of implemented UML class diagram refactorings contains:

1. Extract Superclass;
2. Collapse Hierarchy;
3. {Add, Extract, Remove, Move, Rename} Class;
4. {Add, Remove, Rename, Pull Up, Push Down} Attribute.

These generic, pre-defined refactorings can be used for any GME metamodel. Refactoring strategies for user-defined refactorings can be specified and implemented using a special underlying language, called Embedded Constraint Language (ECL). Users of the refactoring browser are also allowed to customize pre-defined refactorings by modifying the corresponding ECL code. Generally, a refactoring is composed of a name, several parameters, preconditions and a sequence of strategies.

According to Zhang *et al.* [2004], ECL is an extension of OCL providing many of the common features of OCL, such as arithmetic, logical and collection operators. Additionally, it provides special operators supporting model aggregates, connections and transformations (e.g. addModel, setAttribute or removeNode) that can access model elements stored in GME.

3.4.5 Odyssey-PSW

Correa & Werner [2004] discuss how refactoring techniques can be applied in order to improve the understandability and support the evolution of UML/OCL models, i.e. models consisting of UML class diagrams and OCL expressions.

Analogously to "code smells", the term "OCL smell" is introduced and defined as "a hint that some part of an OCL specification or even of the underlying class model should be refactored." Correa and Werner identified and described five the most common OCL smells:

1. Magic Literal – "a numeric or string literal that appears on the middle of an expression without explanation";
2. And Chain – "a single constraint (invariant, precondition or postcondition) composed of two or more expressions connected by *and* operators";
3. Long Journey – "an OCL expression that traverses many associations between different classes of the model";
4. Rules Exposure – "business rules details are specified in the preconditions or postconditions of system-level operations";
5. Duplicated Code – "the presence of duplicate OCL expressions".

The research paper contains also descriptions and examples of a number of UML/OCL model refactorings which are classified into three categories:

1. OCL-exclusive refactorings (affect only OCL expressions):
    a. Add Variable From Expression;
    b. Replace Expression By Variable;
    c. Split AND Expression;
2. UML diagram refactorings (changes in class definitions that may have an impact on OCL expressions) - all refactorings discussed by Sunyé *et al.* [2001] plus renaming refactorings;
3. OCL definition constraint refactorings (changes made to OCL expressions that: introduce new elements in the class definitions or are related to OCL definition constraints):
    a. Add Operation Definition From Expression;

b. Replace Expression By Operation Call Expression.

Analogously to the research carried out by Sunyé *et al.* [2001], supported are only these UML diagram refactorings that are based on rules that can be verified through analysis of structural relationships between model elements. Refactorings that demand more complex semantic equivalence analyses (e.g. involving type casting and polymorphic operations) have been deferred to a future work.

The OCL refactorings described by Correa & Werner [2004] have been defined and automated in a prototype tool Odyssey-PSW (Precise Specification Workbench), which is an add-in to existing OO CASE tools able to access a UML model through a XMI interface. A refactoring is defined as an update operation having: a name, a textual documentation, parameters, pre- and postconditions (defined in OCL) and a body that implements the transformation (in OCL-Script). OCL-Script is an imperative language, similarly to ECL [Zhang *et al.* 2004] based on OCL. It allows, among other things, actions such as: creation and deletion of instances of a MOF compliant metamodel, assignment to attributes and association ends, operation calls, and operations that modify the contents of a collection.

Correa & Werner [2004] propose an interesting, so called *regression animation technique* for checking whether or not the semantics of the model is preserved when a refactoring is performed manually. To be animated by Odyssey-PSW, all query operations in a UML/OCL model must have their bodies defined in OCL and non-query operations – in OCL-Script. Additionally, the designer must specify a set of test cases that define states of the model before and after execution of some of the operation defined in the model. The Odyssey-PSW animation module checks whether any invariant, pre- or postcondition is violated during an animation scenario.

3.5 Vertical Software Consistency

Usually, software is composed of many different kinds of software artefacts, such as requirements documents, analysis models, design models, source code, test suites, etc. Therefore, if any of these software artefacts are being refactored, the others have to be kept consistent.

In this section we will focus on the vertical, as opposed to horizontal, consistency between UML models and code, which maintenance – in the context of software consistency and model refactoring – is the most vexed issue.

Massoni [2003], who investigates introduction of refactoring to heavyweight software methodologies – in particular to the Rational Unified Process – lists three, commonly used in software development, code-model consistency approaches: simple forward engineering, successive reverse engineering and round-trip engineering. Next, he identifies potential problems that may occur during refactoring source code (Java), design models or analysis models and trying to maintain a vertical consistency between them with the use of round-trip engineering.

Most of the problems mentioned by Massoni appear when source code is regenerated from refactored models, another result from difficulties in dealing with different levels of abstraction and implementation specifics. All these issues are caused by the fact that although there is a similarity between UML models and object-oriented code, there is no fixed correspondence between them [Fowler 2003]. Round-trip-engineering bases on two kinds of mappings: code to model and model to code, which, from the mathematical point of view, are not isomorphic, i.e. there occurs an

information loss in the case of both mappings [Rumpe 2002]. The primary intention of UML models is not to represent underlying source code but rather to provide means for object-oriented analysis and design. That is why concepts like invocations and accesses are especially hard to model using UML [Tichelaar *et al.* 2000; Tichelaar 2001].

According to Van Gorp *et al.* [2003b], although model refactorings are expressed at the design level, they must be aware of all the detailed code-level issues. This problem was already noticed by Demeyer *et al.* in a research paper concerning UML shortcomings for coping with round-trip engineering [1999], as well as by Sunyé *et al.* [2001] who provide two examples of refactorings (Move Method and Specialization) which pre- and postconditions – in the absence of information about method bodies of particular operations – cannot be verified at the model level. In the case of Move Method, the body of the concerned operation must not refer to attributes and only navigate through an association to the target classifier. Therefore, the transformation requires information about attributes and methods used inside the body of the method being implementation of this operation. A precondition for the Specialization refactoring states that the *reference context* of a pushed down element (attribute or operation) must not be its owner class. There is no other way to obtain the reference context of an attribute than analyze method bodies of operations from associated classes.

Van Gorp *et al.* [2003b] argue that the UML 1.4 [OMG 2002] metamodel is inadequate for maintaining the consistency between design models and corresponding program code. The UML 1.4 metamodel considers method bodies as implementation specific and therefore, typical UML tools treat them as "protected areas", which must be supplied manually and are not altered during code (re)generation. After refactoring a UML model and next regenerating a source code, it is common that inconsistencies appear in these "protected areas". For example, in the case of Pull Up Method refactoring, a UML 1.4 metamodel based tool must be able to decide on equality of methods, in order to remove from superclasses all copies of a pulled up method. Even in the case of the simple Rename Class refactoring, such a tool is not able to update the refactored class's name in type declarations, type casts and exceptions. On the other hand, given a precise model of statements in a method body, a UML tool would be able to perform even such typical code level refactorings, like Extract Method.

3.6 Source-Consistent UML Model Refactoring

For the needs of this section, let us define a *source-consistent UML model refactoring* as a UML refactoring that maintains consistency between refactored model and underlying source code.

In order to prove that the UML 1.4 metamodel is almost sufficient to allow for expressing source-consistent model refactorings, Van Gorp *et al.* [2003b] carried out an experiment, which goal was to provide concrete suggestions on realization of an ultimate UML refactoring extension. Within the framework of the experiment, they constructed *GrammyUML* metamodel (see **Figure 3.1**), which bases on the UML 1.4 metamodel and includes eight additive extensions that allow for, among others, modelling statements in method bodies and use of typed local variables in a given scope. With the purpose of verifying access-, call- and update-preservation of refactorings (see Section 2.5), several stereotypes have been defined and incorporated into *GrammyUML*, along with four new refactoring Well-Formedness Rules.

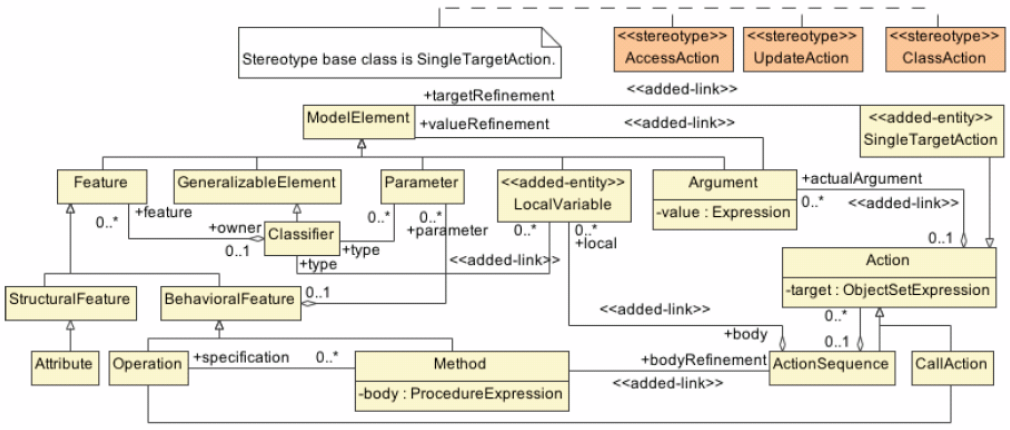

Figure 3.1. Overall view of *GrammyUML* metamodel [Van Gorp *et al.* 2003b]

In the next step, Van Gorp *et al.* [2003b] described in OCL, at the level of *GrammyUML* metamodel, the *refactoring contracts*, i.e. associated bad smells, pre- and postconditions, of two sample refactorings, namely Extract Method and Pull Up Method. As already stated in Section 3.5, it would be impossible to express refactoring contracts of these refactorings at the level of UML 1.4 metamodel.

Van Gorp *et al.* [2003a; 2003c] validated their approach by implementing Pull Up Method refactoring in an open source UML CASE tool called Fujaba, with the use of Story Driven Modeling (SDM), a visual programming language based on UML and graph rewriting. The most straightforward solution would be to use instead of Fujaba an OCL-enabled tool, however they did not do this due to "the practical unpopularity of OCL (both in use by developers as in tool support) [Van Gorp *et al.* 2003a]."

Fujaba's metamodel consists of two layers of abstraction: the first one is equivalent to the UML 1.4 metamodel and the second one refines the method body as a Java abstract syntax tree (AST). However, the second layer, as it does not contain the explicit access, update and call information, is not suitable for reasoning about refactoring. To solve this problem, it was replaced by *GrammyUML* extensions. Unfortunately, also the SDM turned out to be not expressive enough for refactoring purposes. Van Gorp *et al.* bypassed its shortcomings by implementing SDM-inexpressible constraints in Java but because it is not an "elegant" solution, they proposed additionally to extend SDM with parameterisation of graph expressions and story patterns.

In order to ensure appropriate source code regeneration from refactored models, Van Gorp *et al.* suggest introduction of a new component called *Code Preserver* into the Fujaba architecture. They define it as "a development tool component that stores all the required source code files from which a model is extracted in such a way that the complete system can be regenerated from a transformation of the input model [Van Gorp *et al.* 2003c]." The need for *Code Preserver* results mainly from the fact that *GrammyUML* metamodel, since it includes only a minimal set of information sufficient for reasoning about refactoring, does not contain all syntactically possible source code constructs.

3.7 Executable UML Model Refactoring in AGG

According to Massoni [2003], other – alternative to round-trip engineering – approaches addressing code-model consistency issue are (1) attaching source code to

models, (2) recording refactorings, and (3) executable UML. The second approach can be realized by e.g. describing refactorings as coordinated graph transformation schemes that are instantiated according to the specific code modification and applied to the design models affected by the change [Bottoni *et al.* 2003]. The last method, i.e. executable UML (for details see Section 4) seems most promising since in this approach, as the models are automatically translated into executable entities, the problem of vertical code-model consistency does not exist.

To our knowledge, so far there is only one research paper that concerns refactoring of executable UML models, namely "Formalizing Refactoring by Using Graph Transformation", published by Kazato *et al*. in 2004. The authors of this article deal with refactoring of design models comprising of both structural and behavioural parts. The former is expressed by classes and related structural concepts, and the latter is specified with the use of operation bodies implemented in action semantics.

Kazato *et al*. [ibid.] defined a set of twenty-eight *basic transformations* of design models, of which various refactorings can be composed. For example, the Push Down Attribute refactoring (derived from Fowler's Push Down Field) consists of two basic transformations, namely *push down attribute to the subclass*, and *remove attribute*. As in all other studies, refactoring preconditions are specified with the use of OCL.

The approach to implementation of model refactorings proposed by Kazato *et al.* [ibid.] bases on an observation, that each UML model may be represented as an object model comprised of instances of metaclasses of the UML metamodel. A part of an exemplary model and its so-called *repository-view* [Bock 2003] are shown in **Figure 3.2**. Each such an object model is essentially a typed and attributed directed graph, and therefore each basic model transformation can be treated as a graph transformation with rules described as a part of the graph grammar.

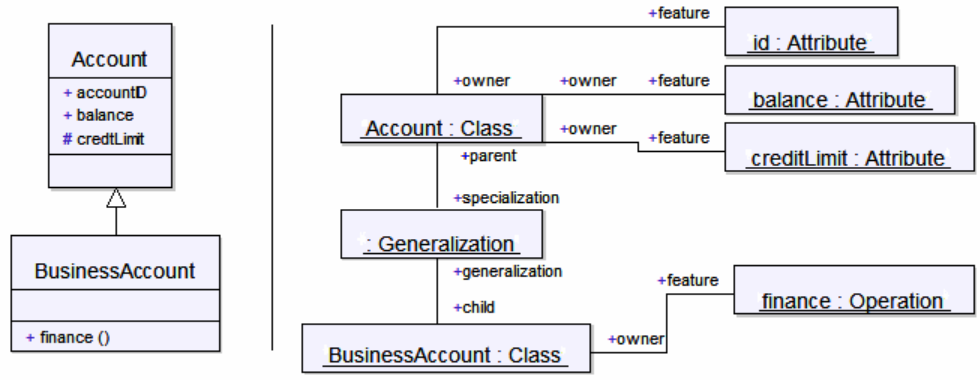

Figure 3.2. Repository-view (right) of a simple UML model (left) [Kazato *et al.* 2004]

From the implementation-viewpoint, i.e. when considering the repository representation of the model, basic transformations can be further decomposed. In our previous example, *push down attribute to the subclass* consists of *create attribute* and *create association*, executed for each subclass.

Kazato *et al*. [ibid.] successfully implemented six[10] compound refactorings as well as their preconditions, in the form of twenty-five OCL queries, in a graph transformation system called Attribute Graph Grammar (AGG).

[10] Mentioned are only: Self Encapsulate Attribute, Remove Setting Method, and Extract Class.

4 EXECUTABLE MODELLING WITH UML

The main goal of this chapter is to present a literature survey on executable modelling with UML. It is structured as follows: Section 4.1 situates executable UML against a background of the evolution of software engineering; Section 4.2 identifies an executable subset of UML 2.0; Section 4.3 provides an overview of several exemplary attempts to achieve executable UML. Finally, Section 4.4 concludes this chapter with a discussion on advantages and disadvantages of UML as a programming language.

4.1 Introduction

Grady Booch, the chief scientists for former Rational Software, said in an interview for CHIPS Magazine "the history of software engineering has been one of growing levels of abstraction (…) This growth has occurred simply as a meaningful engineering response to the growth in complexity in the kinds of software systems we as an industry are asked to create [Booch 2002]." The complexity of software systems can be measured e.g. using function points (FP) that are a unit measure for software that quantifies its functionality provided to the user basing primarily on the logical design [LC 2004]. According to Longstreet Consulting Inc., [LC 2004a] the size of large projects increased 10 times between years 1970 (1000 FP/project) and 2000 (10000 FP/project).

The evolution of software engineering, although present in development methods and software platforms, is most clearly apparent in the history of programming languages. Richard M. Soley wrote in the foreword to a book on MDA [Mellor *et al.* 2004] that a "critically important" first step in this evolution was, developed in 1954 and released in 1957, FORTRAN language (FORmula TRANslation system). However, one should bear in mind that this evolution began even earlier [Hightower 1996]. First programmers had to load programs manually into memory, and after their completions unload them, using switches and buttons. The invention of operating systems automated this process and the introduction of assembly languages allowed developers to program computers without worrying about the correctness of used instructions.

The development of FORTRAN not only accelerated the process of programming but also, due to the application of compilers, enabled portability of programs that could be finally written once and automatically translated for different machines. In this way, independence from hardware platforms has been achieved, and "since then, we have apparently flown up the abstraction ladder [R.M. Soley in Mellor *et al.* 2004]." Later languages, offering e.g. automatic memory management, let programmers focus more and more on application domain instead of on solution domain, what significantly increased their productivity. In the same time, the biggest problem became software portability across different operating systems. Programming languages operated on abstraction level that turned out to be too low to be able to write an application that could be directly compiled and run on e.g. Microsoft Windows and UNIX, not mentioning MacOS or any of the mobile phone operating systems. This shortcoming was the main reason for introduction of software platforms like CORBA, J2SE, J2EE or .NET that are independent from underlying operating systems. Additionally, they provide mechanisms for, among others, transparent distribution, concurrency and

persistency, what again soars programmers productivity, which increased between years 1970 and 2000 more than 3 times [LC 2004a].

Did the evolution reach its end? Boyd [2002] gives a negative answer to this question. In his opinion, the next logical, and perhaps inevitable, evolutionary step is *executable UML* [ibid.]. According to Pender, the term executable UML is used to describe "the application of a UML profile in the context of a method that aims to automatically generate an executable application from an abstract UML model [2003]". Ivar Jacobson states in the foreword to a book on Executable UML[11] (xUML) that the software community for a long time aimed at creating a modelling language that is also an executable one [Mellor & Balcer 2002]. He notices analogy between UML and another modelling language, namely Specification and Description Language (SDL), which was crated in the International Telecommunication Union (ITU). In the early 1980s, SDL has been extended by constructs to formally define algorithms and data structures, and thus evolved into a high-level programming language. However, neither SDL was the first one, since it was inspired by its (and UML's) forerunner, so called "The Ericsson Language", created already in 1968. It served to model telecommunication components with the use of sequence diagrams, collaboration diagrams and state transition diagrams (a combination of statechart and activity diagrams), and it enabled generation of almost 90% of the source code.

4.2 Executable Subset of UML 2.0

Before answering a research question, whether UML has potential for being a programming language, one has to ensure that it can be executed. The IEEE standard 610.12-1990 defines computer program as a "combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions [IEEE 1990]." One can therefore assume that an executable model of a system shall consist of a specification of its structure (data definitions) and behaviour (computer instructions).

According to Rumpe [2002], already in UML 1.3 there is a large subset of modelling constructs, which can be animated and therefore used to specify as well executable models as test cases for them. This subset consists of class diagrams, object diagrams, statechart diagrams, activity diagrams, sequence diagrams, collaboration diagrams and OCL constraints. Sequence or collaboration diagrams are suitable to specify test cases, and statechart or activity diagrams are appropriate to describe behaviour of a single object. Class and object diagrams along with OCL constraints can be used as an input to as well object code as test code generation.

4.2.1 General Information on UML 2.0

UML 2.0 specification consists of four parts, namely:

- *UML 2.0 Superstructure* [OMG 2004] – it defines the user level constructs;
- *UML 2.0 Infrastructure* [OMG 2003] – it defines the foundation language constructs for both UML 2.0 superstructure and MOF 2.0;
- *UML 2.0 Diagram Interchange* [OMG 2003a] – it defines a supplementary package for graph-oriented information, enabling a smooth and seamless exchange of models between different software tools.

[11] It should be noted the difference and relation between *executable UML* and *Executable UML (xUML)*, namely *xUML* is one of the approaches to achieve *executable UML.*

- *UML 2.0 Object Constraint Language (OCL)* [OMG 2003b] – it defines a formal language used to specify expressions on UML models;

Adoption of the UML 2.0 Superstructure is complete, and so-called Available Specifications of other three parts of UML 2.0 are supposed to be posted for non-OMG members in the middle of 2005.

According to Kobryn [2004], the major improvements in UML 2.0 are:

1. *Support for component-based development via composite structures* – both Classes and Components can be decomposed and assembled via Parts, Ports, and Connectors;
2. *Hierarchical decomposition of structure* (Classes and Components) *and behaviour* (e.g. Interactions, State Machines, Activities);
3. *Cross integration of structure and behaviour* – the same model element can be used in different kinds of diagrams;
4. *Integration of action semantics with behavioural constructs* – UML actions are defined in as much detail as a programming language's statements;
5. *Layered architecture to facilitate incremental implementation and compliance testing* – UML concepts are contained in Packages, which are in turn partitioned into four horizontal layers of increasing capability called *compliance levels*.

4.2.2 Run-Time Semantics of UML 2.0

Run-time semantics of UML 2.0 [OMG 2004; Selic 2004] are specified as a mapping of modelling concepts into corresponding program execution phenomena. They are based on two fundamental premises:

1. All behaviour in a modelled system is caused by actions executed by so-called *active* objects;
2. UML behavioural semantics deal only with *event-driven* (discrete) behaviours.

The *causality model* of UML 2.0, i.e. a "specification how things happen at run-time [ibid.]", can be summarised as follows – objects respond to messages sent by the ones that execute communication actions, by executing behaviours attached to these messages. An exemplary scenario[12] is depicted in a communication diagram in **Figure 4.1**. The example shows two independent and possibly concurrent threads of causally chained interactions. The first, identified by the thread prefix 'A', consists of a sequence of events that begin with `activeObject1` sending signal `s1` to `activeObject2`. In turn, `activeObject2` responds by invoking operation `op1()` on `passiveObject1` after which it sends signal `s2` to `activeObject3`. The second thread, distinguished by the thread prefix 'B', starts with `activeObject4` invoking operation `op2()` on `passiveObject1`. The latter responds by executing the method that realizes this operation, in which it sends signal `s3` to `activeObject2`.

[12] This example is taken directly from the Superstructure specification [OMG 2004].

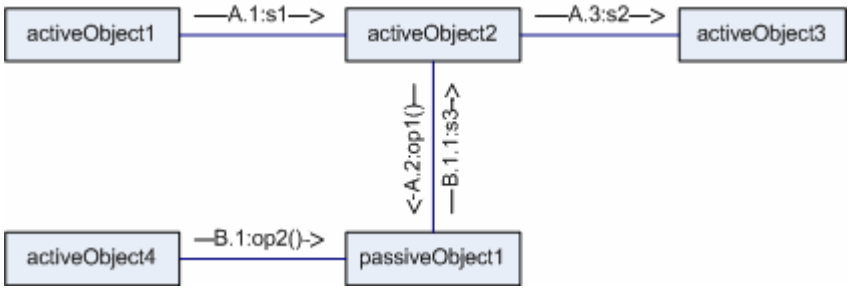

Figure 4.1. Example illustrating the basic causality model of UML 2.0 [OMG 2004]

The key semantic areas covered by UML 2.0 and the relation between them are presented in **Figure 4.2**. At the highest level of abstraction, the architecture of UML 2.0 run-time semantics consists of three composite layers[13]: (1) Structural Foundations, (2) Behavioural Base, and (3) High-Level Formalisms.

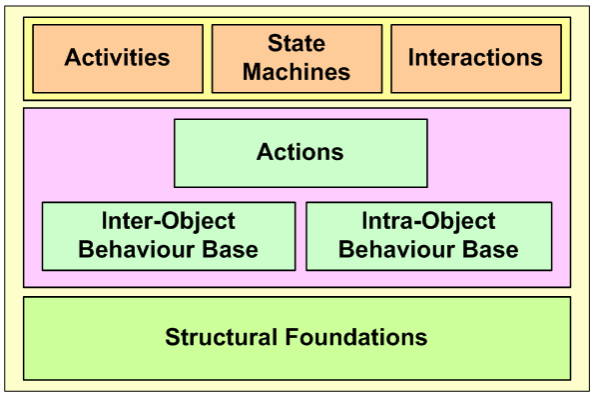

Figure 4.2. The architecture of the UML 2.0 run-time semantics [OMG 2004]

The first layer reflects the premise stating that in UML there is no disembodied behaviour. The second layer provides the base for the semantic description of all the higher-level behavioural formalisms. It consists of three separate sub-areas arranged into two sub-layers. The bottom sub-layer is composed of the *inter-object behaviour base*, which concerns communication between structural entities, and the *intra-object behaviour base*, which addresses the behaviour occurring within them. The *actions* sub-layer, described in the sequel of this section, defines the semantics of individual actions. The topmost layer in the architecture defines the semantics of the following high-level behavioural formalisms of UML: *activities*, *state machines*, and *interactions*.

A more detailed explanation of the two bottom layers can be found in a paper written by Selic [2004]. Obviously, a thorough description of the whole run-time semantics of UML is given in the UML 2.0 Superstructure specification [OMG 2004]. Therefore, this section contains only a presentation of the core *actions* sub-layer.

According to Pender [2003], the primary reason for having action semantics is to provide a standard for the exchange of action specifications between tools. On the contrary, Rumpe [2002] states, that the main reason for extending UML by an action language is the shortcoming to generate code from OCL constraints. Nevertheless, integration of actions with behavioural constructs is – from the viewpoint of model execution and simulation – the most significant improvement in UML 2.0.

[13] The items in the upper layers depend on the items in the lower ones.

As defined by the UML 2.0 standard, an action is "the fundamental unit of behaviour specification [OMG 2004]", that converts a set of inputs into a set of outputs. The actions are organized into four packages, which are shown – together with their inter-dependencies – in **Figure 4.3**. *BasicActions* package is required at the Compliance Level 1, *StructuredActions* and *IntermediateActions* – at the Level 2, and *CompleteActions* – at the Level 3.

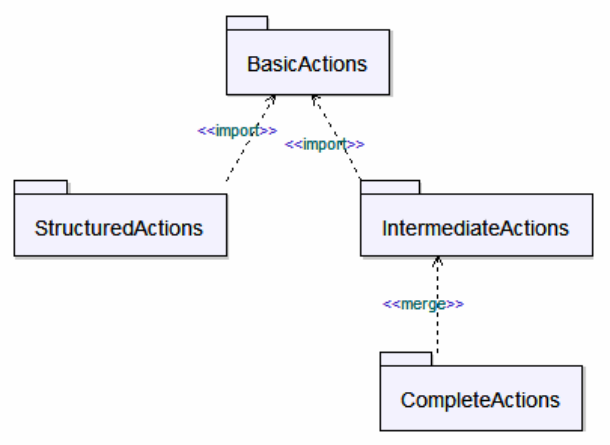

Figure 4.3. Inter-dependencies of the Actions packages [OMG 2004]

BasicActions contain various actions that invoke behaviour, namely *SendSignalAction* that creates a signal instance from its inputs and transmits it to a target object, *CallOperationAction* that transmits an operation call request to a target object, and *CallBehaviorAction* for direct behaviour invocations. Additionally, the basic package includes *OpaqueAction* with implementation-specific semantics.

The characteristic feature of intermediate actions is that they either carry out a computation or access object memory. *IntermediateActions* contains two additional invocation actions, namely *BroadcastSignalAction* that transmits a signal instance to all the potential target objects in the system, and *SendObjectAction* for transmission of objects. Another set of intermediate actions constitute so-called object actions, i.e. *CreateObjectAction*, *DestroyObjectAction*, *TestIdentityAction* to test if two values are identical objects, and *ReadSelfAction* that retrieves its host object. The package contains four actions that concern structural features – *ReadStructuralFeatureAction*, *AddStructuralFeatureValueAction*, *RemoveStructuralFeatureValueAction*, and *ClearStructuralFeatureAction* that removes all values of a structural feature. Link actions constitute another group of intermediate actions – these are *ReadLinkAction* that navigates across associations to retrieve objects on one end, *CreateLinkAction*, *DestroyLinkAction* that destroys links and link objects, and *ClearAssociationAction* for destroying all links of an association in which a particular object participates. Additionally, the intermediate package includes *ValueSpecificationAction* that returns the result of evaluating a value specification.

StructuredActions contains *RaiseExceptionAction* that causes an exception to occur, and variable actions organized similarly to structural feature ones (from *IntermediateActions*) – these are *ReadVariableAction*, *AddVariableValueAction*, *RemoveVariableValueAction*, and *ClearVariableAction*.

One of the groups of actions from *CompleteActions* deals with accepting events, and it constitutes of *AcceptEventAction*, *AcceptCallAction*, *ReplayAction*, and *UnmarshallAction*. The package contains four object actions, namely

ReadExtentAction that retrieves the current instances of a classifier, *ReclassifyObjectAction* that changes a classifier of an object, and *StartClassifierBehaviorAction* that provides a way to indicate when the execution of the classifier behaviour of a newly created object should begin. Link object actions from *CompleteActions* operate on instances of association classes – these are *CreateLinkObjectAction*, *ReadLinkObjectEndAction* that retrieves an end object from a link object, and *ReadLinkObjectEndQualifierAction.*

Additionally, the UML standard defines two actions for dealing with time values, namely *TimeObservationAction* that observes the current point in time, and *DurationObservationAction* that observes duration in time. Both these actions come from *CommonBehaviors::SimpleTime*, and they write values of their observations to structural features.

4.3 Attempts to Achieve Executable UML

As indicated by Mellor & Balcer [2002], there are many executable subsets of UML. The goal of this section is to provide an overview of several exemplary as well commercial as research attempts to achieve executable UML. Description of each approach focuses particularly on two aspects, namely (1) what means can be used to specify a model, and (2) how is the model executed.

4.3.1 Combining UML with Programming Languages

Combining UML models with fragments of code written in one of the contemporary programming languages (e.g. C++ or Java) is the most straightforward attempt to achieve an executable UML model. This approach merges two distinct activities of design and implementation into one, which is usually followed by source code generation and testing.

The metamodel of e.g. UML 1.4 [OMG 2002] contains several places in which UML can be mixed with programming languages. One of the examples is *ProgrammingLanguageDataType*, which can be used to capture type constructs not included as UML classifiers. Another example is *ProcedureExpression* – according to the abstract syntax for the Core package of UML 1.4, the type of *body* attribute of the *Method* metaclass, i.e. an implementation of method's body, is a *ProcedureExpression*, which "defines a statement that will result in a change to the values of its environment when evaluated [OMG 2002]". Such statements can be therefore written in any of programming languages.

4.3.2 Executable UML (xUML)

Executable UML (xUML) [Starr 2002a; Mellor & Balcer 2002; Mellor *et al.* 2004] is a profile of UML that defines execution semantics for a computationally complete subset of this language[14]. It originates from Shlaer-Mellor community and it can be used to build Platform-Independent Models (PIMs) that make no decisions about a particular hardware and software environment. According to Ivar Jacobson [Mellor & Balcer 2002], xUML is one of the cornerstones on which rests the Model Driven Architecture (MDA) initiative.

[14] For an open source, and unfortunately incomplete version of the xUML metamodel see [Starr 2002b].

A complete xUML specification of a system consists of a number of autonomous, reusable, and replaceable domains, which aggregate sets of entities modelled with the use of UML classes, which in turn may have lifecycles (behaviours over time) that are abstracted as state machines. The behaviour of the system is driven by objects moving from one stage in their lifecycles to another in response to events. Each state machine has a set of procedures, one of which is executed when the object changes state, thus establishing the new state. Each procedure comprises a set of actions, being primitive units of computation, which cause e.g. synchronization or data access to be executed. In the matter of actions, xUML relies on the *UML 1.4 with Action Semantics* [OMG 2002a] and allows using any surface language that is compliant with this specification.

The classes and their relationships are illustrated on class diagrams, and their state machines with procedures embedded in states are shown on statechart diagrams. Other kinds of diagrams, i.e. collaboration diagrams and sequence diagrams, can be automatically generated from xUML models. What mostly distinguish xUML from the ordinary UML are operations, which are derived from actions on state machines. Moreover, neither aggregation nor composition relationships are supported.

The next step after creation of complete models of domains is determination of how these models are supposed to be linked together, especially which identifiable entities in one model correspond to other ones in another models. This activity is called *bridging domains,* and it can be performed in both explicit and implicit style [Mellor & Balcer 2002]. The source models can be optionally *coloured* with performance and deployment decisions [Starr 2002a] called in the MDA terminology *marks* [Mellor *et al.* 2004]. Afterwards, a model compiler weaves together the models according to a single set of architectural rules, so called *archetypes*, i.e. fragments of data access and text manipulation logic that state formally how to translate an xUML model into text being e.g source code written in Java, C++, VHDL, or COBOL. Besides weaving and translating all source models into code, the model compiler must also incorporate elements than enable among others storing instances, generating calls and signals across task and processor boundaries, and traversing state machines [ibid.]. All this elements constitute a xUML execution engine targeted to a selected software platform.

4.3.3 UML Virtual Machine

Schattkowsky & Müller [2004; 2004a] present an approach for model-based development of embedded systems applying a well-defined UML 2.0 subset with precise execution semantics, and support for timeouts as well as exception and interrupt modelling.

The structure of systems built in this approach can be specified with the use a subset of UML 2.0 *Classes* package, which encompasses among others classes having attributes and operations (called only synchronously), support for single inheritance as well as realization of multiple interfaces.

The behaviour of each non-abstract operation is modelled with the use of a state machine, and each activity of a state or a transition in these state machines is specified via an interaction, illustrated in a sequence diagram. Such a combination of state machines and interactions allows overcoming the limitations of state machines when expressing complex algorithms and deeply nested control flows [ibid.].

A state machine consists of start and final pseudo states, as well as of simple and composite ones, where each latter contains another state machine. Transitions between states can be triggered by among others an occurrence of a timeout or hardware interruption, a software exception, i.e. division by zero, or an explicit trigger from the current state's implementation. The whole operation completes when the associated state machine terminates by reaching a final state.

The surface syntax of the language used to specify single actions is derived from C++ and Java. The language allows value assignments based on nested expressions and operation invocations using variables in the current scope [ibid.]. As well basic math and bit as logic operations are covered, and their semantics is comparable to the one of their Java equivalents.

The complete UML specifications are automatically transformed to equivalent executable state-oriented models, consisting of binary finite state machines (obtained from UML state machines) with byte code (obtained from UML interactions) embedded in states. The mapping between input and output model elements is not of the one-to-one type – e.g. entry-, exit-, and transition-activities in input state machines are transformed to additional states and transitions in output ones. Such a semantics-preserving simplification reduces the number of elements of which the execution environment has to be aware. After the transformation, the output models can be executed directly by a dedicated UML Virtual Machine (UVM), implemented in either hardware or software.

Riehle *et al.* [2001] provide the description of both logical and physical architecture of another UML Virtual Machine, which is implemented in Java as an object-oriented framework that has ability to execute models by interpreting them according to UML semantics. Models are built with inter-related classes, and their behaviour is specified by state charts reacting to events that they receive. Class descriptions are enriched by OCL constraints that represent the inter-object dependencies resulting from among others business rules – this way, state transitions in one object are translated into events relevant to other objects that are not connected with the originating object through a state chart. Unfortunately, since models created for this virtual machine are based on UML 1.3, detailed operational behaviour has to be implemented manually using native Java, what is doubtlessly the major drawback of this immensely interesting approach.

4.3.4 Comparison of the Attempts

The goal on this section is to introduce a coarse categorisation of approaches to achieve executable UML, as well as indicate their major strengths and weaknesses.

Basing on several exemplary, as well commercial as research solutions described in literature, two main variation points of the approaches have been identified, namely *system specification* and *model execution*. In the context of *system specification*, it has been discovered that (1) in all the attempts, structural aspects of systems are modelled with the use of classes, and (2) different subsets of UML along with either programming languages or action semantics are used to model systems' behaviour. From the *model execution* perspective, all the approaches fall into two broad categories of (1) platform-specific code generation, and (2) execution on a virtual machine.

Feng [2003] mentions several disadvantages of combining UML with programming languages, namely:

- No programming language contains all the useful software concepts;

- Not all UML concepts are directly supported by contemporary programming languages;
- Using a programming language enforces to focus on implementation details too early;
- Models created in this way are not portable across different modelling tools.

There are two kinds of platform-specific code generation, namely *naive code generation* that is usually preceded by specification of system's behaviour using a programming language, and more sophisticated *model compilation*. In the former approach, unchanged language-specific expressions and statements are naively placed in the source code generated from UML specification of system's structure, where in the latter one, a model compiler is aware of syntax and semantics of an action language used to express system's behaviour.

The major drawbacks of the naive code generation are [Feng 2003]:

- Modellers are not able to test the system until the source code is finally generated;
- Modellers must have a comprehensive knowledge of code generation rules;
- Automatic model analysis and verification is difficult, what makes the generated software error-prone and hinders debugging;
- The process of code generation is time-consuming, what significantly increases system's time-to-market.

Model compilation approaches are free from above-mentioned weaknesses, because model verificators and simulators usually accompany development environments supporting model compilation. However, disregarding the way the source code is obtained, the process of its generation is time-consuming, what significantly increases system's time-to-market. Moreover, the time delay caused by code generation, its compilation, shutting down the existing system, installing and configuring the new one and starting it up, makes simulation of new models with immediate user feedback uncomfortable if not impossible. The resulting models easily become not optimal [Riehle *et al*. 2001].

As indicated by Schattkowsky & Müller [2004], the use of a virtual machine eliminates the need to compile models to different platforms – instead, only the virtual machine itself needs to be ported to each platform. As a model is supposed to run on any VM implementation, improvements to the runtime environment are immediately beneficial for existing software, what also significantly reduces cost for application development and testing.

Platform-specific code generation, in its two variants, is a much more common and mature technique for execution of UML models that the use of a virtual machine. Actually, all commercial tools that were investigated by the authors support this approach and only few research projects use the latter one.

Furthermore, with respect to behavioural specifications, models can be categorized into *state-oriented* and *stateless*. In the state-oriented paradigm, a structural entity may have a state machine that describes its lifecycle. On the contrary, stateless models are very similar to programs written in object-oriented languages.

The attempts to achieve executable UML that are described in Section 4.3 are of course not the only ones. Among others, there are several commercial UML 2.0 based

tools[15], which enable creation and execution of models, like I-Logix Rhapsody[16] [Niemann 2004] or Telelogic TAU[17] [Björkander & Kobryn 2003; Kobryn & Samuelsson 2003; Leblanc 2004; Telelogic 2004]. A detailed description of TAU executable models, which can be treated as a supplement to this chapter, can be found in Section 5.

4.4 UML as a Programming Language

Being executable is an essential, but not a sufficient condition that UML has to meet in order to replace currently used high-level programming languages. Rumpe, in a paper on UML in the context of extreme modelling [2002], poses on UML six following requirements:

1. UML needs to be fully expressive,
2. UML needs to be more compact notation than an ordinary programming language,
3. UML needs a simple and usable module concept,
4. UML needs support for testing,
5. UML needs an adequate tool support.
6. UML needs an effective translation into efficient code,

In fact, only four first requirements concern UML as a language. The state-of-the-art UML 2.0 seems to fulfil them, as it (1) includes a wide set of actions that make it fully expressive, (2) is more compact than programming languages by disregarding implementation (platform) details, (3) has a simple and usable module concept in the form of a package, and (4) contains interactions that are particularly suitable for testing purposes. The adequate tool support for executable UML and its effective translation into efficient code are just the matter of time.

4.4.1 Advantages of Executable UML

Four interrelated categories of advantages of programming directly in the modelling language have been identified, namely the ones resulting from:

- the elimination of the *two-language problem;*
- the platform- and target-language-independence;
- the compact and graphical notation of UML;
- the early model execution and automatic code generation capabilities.

One of the advantages resulting from the elimination of the *two-language problem*, i.e. modelling in UML, and programming in e.g. C# or Java [Jacobson in Mellor & Balcer 2002], is **improved communication**. Analysts, designers, programmers, and testers all use the same language, what "reduces the differences between their roles in the development process and helps to remove natural barriers that moving from the model domain to the code domain inadvertently introduces [Björkander 2000]." Next advantages in this category are **reduced number of artefacts**, and **no need for code-model synchronization**, which both result from the fact that the model and the code are essentially the same [ibid.].

The second category encompasses benefits resulting from the platform- and target-language-independence of executable UML models. One of these advantages, called

[15] A list of several UML 2.0 based tools can be found at http://www.uml.org/#Links-UML2Tools.
[16] http://www.ilogix.com/rhapsody/rhapsody.cfm
[17] http://www.telelogic.com/products/tau/tg2.cfm

by us **one model – multiple implementations**, relies on the fact, that an executable model can be interpreted or compiled – without altering it – to code in any desired programming language running on any target platform [Björkander 2000, Björkander & Kobryn 2003]. Modellers that use executable UML are supposed to focus only on what their system should really do. They are allowed to disregard as well certain details of the implementation (e.g. how should an association be implemented?) as other architectural considerations (e.g. concerning distribution – should we use CORBA or COM?) that can be handled later in the development process or left a compiler or code generator. This benefit can be called **focus on functionality**.

As indicated by Soley [Mellor *et al.* 2004], the compact and graphical notation of executable UML, as opposed to textual programming languages, allows the construction of computing systems from models that can be understood very quickly and deeply. This causes another benefit of using executable UML, namely **easier and cheaper software maintenance**. In addition, the tests can be described in a more compact way, what gives rise for **specification-based testing** [Rumpe 2002], which can be even more facilitated by tools that graphically show where a test situation is violated [ibid.].

The fourth category covers advantages resulting from the early model execution and automatic code generation, which render possibility of **model-level debugging** [Björkander 2000], **short development cycles**, and **early feedback for developer** that experiences his model's actual behaviour [Rumpe 2002]. Furthermore, generating code from a model gives a **distinctive time and market advantage** [ibid.].

4.4.2 Doubts Concerning Executable UML

As observed by Björkander [2000], accepting that a modelling language can be used as a programming one may encounter mental hurdles – a common reaction is mistrusting the code that is generated from models. Another doubtful matter is the efficiency of the automatically produced code. However, the same issues raised doubts in the context of e.g. FORTRAN language, which turned out to produce code that was reliable and nearly as efficient as the one written by good programmers. This proves that somehow the high level of abstraction offered by a language does not always have significant run-time costs [Soley in Mellor *et al.* 2004].

According to Fowler [2003], it is worth using the UML as a programming language only if it results in a significant productivity gain when compared to current object-oriented programming languages. However, even if it is more productive, "it still needs to get a critical mass of users for it to make the mainstream [ibid.]." Otherwise, executable UML might meet the same fate as Smalltalk programming language that, despite being very productive, is currently a niche one.

The main doubt seems to be raised by tools, which currently are not mature enough to generate multi-language and multi-platform business applications. Ambler [2003] points out that currently (1) it is difficult to integrate a collection of tools to support executable UML, and that (2) an opposite approach, in which as well a modelling subsystem as code generators are delivered by a single vendor, will likely prove too narrow. As maintained by Ambler [2003], "the complexity of software development and the pace of technological change will outstrip the ability of tools vendors to generate reasonably efficient source code." During several nearest years, it will turn out whether it is true or not.

5 TELELOGIC TAU EXECUTABLE UML MODELS

The goal of this chapter is to provide an overview of executable UML models that can be built and executed in Telelogic TAU. It is structured as follows: Section 5.1 presents the main features of the tool; Section 5.2 describes elements that a TAU executable model must be composed of; Section 5.3 presents an exemplary model built in the tool; Section 5.4 discusses TAU's compliance with UML 2.0. Finally, Section 5.5 presents allowed means of communication between classes in TAU models.

5.1 Introduction

Telelogic TAU Generation2 is the state-of-the-art family of advanced systems and software development and testing tools. It consists of four products directed towards different users, namely:

- *TAU/Architect* - for systems architecture and design,
- *TAU/Developer* – for model-driven software development,
- *TAU/Tester* – for systems and integration testing,
- *TAU/Logiscope* – for software quality assurance and metrics.

In the context of model execution, the highest capability offers TAU/Developer – a UML 2.0 based tool that enables designing, debugging and delivering advanced software components and applications. It enables to:

- create precise visual definitions of software behaviour with the use of a comprehensive action language,
- optimize, compile (to C, C++ or Java) and execute detailed design models,
- graphically trace a running model using sequence diagrams,
- examine a model's behaviour by step-by-step debugging through UML state machines,
- record and rerun model execution steps for future regression testing.

The behaviour of a UML model and its implementation may be verified with the use of the Model Verifier. First, the Application Builder generates an executable program in the C language from the model linked with a predefined run-time library customized for simulation purposes. Next, the program is executed – either automatically or manually, i.e. in a step-by-step manner using various commands and breakpoints. The execution of the session can be traced graphically in state machine and sequence diagrams or textually in the output console window. If the application communicates with the environment, this also may be simulated by sending manually prepared signals.

For the sake of the thesis, Telelogic TAU has been chosen to be the tool, in which selected refactorings are automated. The justification of this choice can be found in Section 6.5.

5.2 Basic TAU Executable Models

In order to be executable with the Model Verifier, a UML model must have a certain level of completion. It must be composed of:

- a package (optional),
- a class diagram (optional),
- at least one active class (the so-called top-level active class),
- at least one state machine with an implementation (optional – in the sense that it can be implicit).

An active class, i.e. a class with its own thread of control, must have a port to be able to communicate with other active classes and/or its environment with the use of signals. A port – a named interaction point of an active class – is defined by the signals, usually encapsulated in interfaces, which it can transmit. A model with (internal or external) communication has:

- at least one signal,
- at least one port,
- an interface (optional).

The lifecycle of an active class is described with a state machine – named *initialize* or having the same name as its owner. The implementation of the state machine is visualized on a statechart diagram (alternatively – in a text diagram). There are two different styles of drawing statechart diagrams supported (see **Figure 5.1**) – the state-oriented view and the transition-oriented one. The first one gives a good overview of a complex state machine but is less practical when focusing on the control flow and communication aspects of a specific set of transitions. For this reason, it is also possible to describe a state machine in the transition-oriented way, with explicit symbols for different actions that can be performed during the transition.

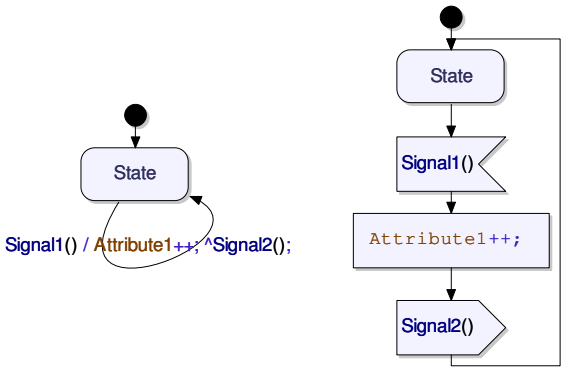

Figure 5.1. State-oriented (left) and transition-oriented (right) syntax of a state machine

Each active class may have its internal run-time structure defined in terms of other active classes, referred to as parts. A composite structure diagram may be used to visualize this architecture as well as to express the communication within an active class by showing connectors between the ports of the parts. A special kind of ports, namely behaviour ports, may be used to enable the communication between a part and the state machine in an instance of the class that owns the part.

More information on TAU executable models in the form of a list of unsupported UML constructs can be found in Appendix E.

5.3 Exemplary Model – Counting Server

The simple exemplary model outlined in this section – Counting Server – is a modified and extended version of one of the projects – Echo Server – supplied with

TAU. The externally observable functionality of the system can be summarized as follows: an actor of the system sends into it two values - an integer number and a time interval – and receives an integer value being either the number multiplied or divided by two – depending on the value of the number.

Figure 5.2 shows a class diagram illustrating the top-level active class of the system – *Server* and two signals – *Count* and *Reply*. *Server* has a port called *EnvPort* used for communication with its environment. The port realizes the *Count* signal, i.e. it declares that *Server* can handle receipts of this one. Moreover, it requires *Reply* signal, i.e. it expects that an actor of the system can handle this one.

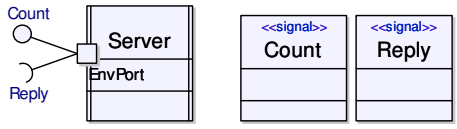

Figure 5.2. External view of the Counting Server

Two latter active classes that fulfil the functionality of the system are *Dispatcher* and *RequestHandler*. The first one has a port called *DPort* that realizes *Count* and requires *Confirm* signals. *RequestHandler* may communicate via its *RHPort*, but only in one direction. Additionally, there is a composition association between *RequestHandler* and a passive class *RequestProcessor*, which implies that each newly created instance of the former one has an attribute *processor* containing an instance of the latter one. All these three classes as well as an additional signal *Confirm* are depicted in a class diagram shown in **Figure 5.3**.

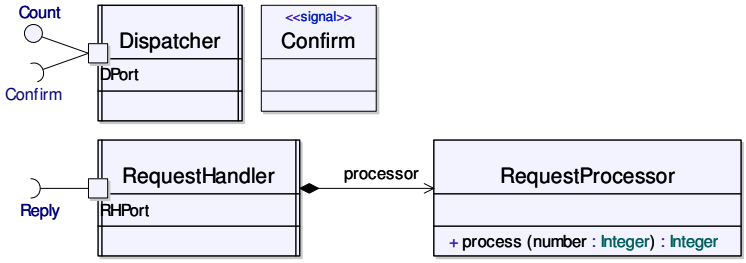

Figure 5.3. Additional classes of the Counting Server

Server has its internal structure depicted on a composite structure diagram shown in **Figure 5.4**. This is the so-called white-box, as opposed to black-box (see **Figure 5.2**), view of the system [Björkander & Kobryn 2003]. It reveals that the active class *Server* has two parts – *d* and *rh* typed by *Dispatcher* and *RequestHandler* respectively. It is noteworthy that the initial number of instances of *RequestHandler* is zero, what indicates that probably they will be created dynamically during execution of the system. Additionally, the composite structure diagram illustrates communication paths – connectors – between ports of all active classes of the system. Therefore, *d* may receive *Count* signals from the environment of *Server* via *EnvToD* connector. Moreover, it can communicate with *Server's* state machine by sending *Confirm* signals to an unnamed behaviour port. Finally, *RequestHandler* may send *Reply* signals to the environment using *RHToEnv* connector.

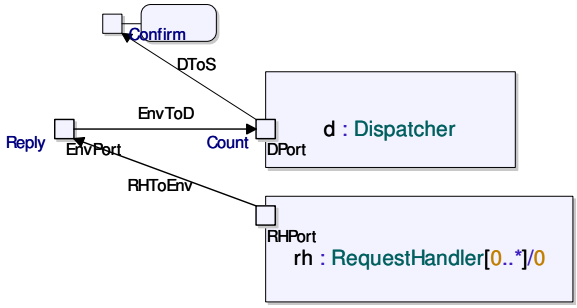

Figure 5.4. Internal structure of *Server* class

Behaviour of the Counting Server is specified with the use of simple state machines that define lifecycles of the active classes.

Server (see **Figure 5.5**) remains in *Idle* state until it receives a parameter less *Confirm* signal that triggers a transition back to the same state. An action that is performed on the transition increments by one the value of an integer attribute *noOfRequests*.

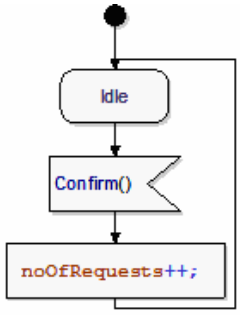

Figure 5.5. State machine of *Server* class

Dispatcher (see **Figure 5.6**) responds to *Count* signal by creating a new instance of *RequestHandler* and sending *Confirm* signal to *Server*. The state machine has two temporary variables – *number* and *pause* – that are used to receive the values carried by *Count* signal and pass them to an instance of *RequestHandler*. It is noteworthy that a constructor of *RequestHandler* is not an operation but a state machine.

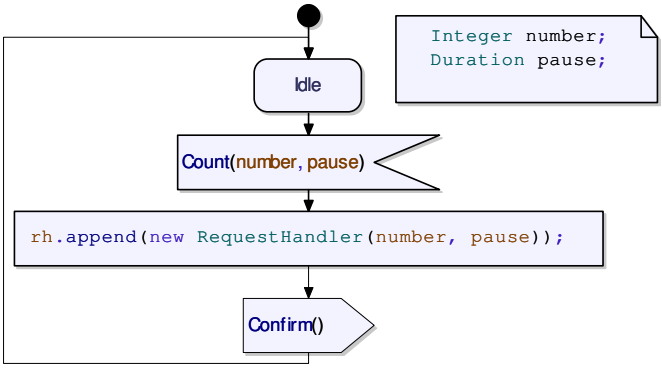

Figure 5.6. State machine of *Dispatcher* class

An instance of *RequestHandler* (see **Figure 5.7**) starts its lifecycle by creating a timer called *PauseTimer* and setting it to trigger a transition after a period determined by the pause parameter. As the transition is triggered, *RequestHandler* invokes *process*

operation of an associated *RequestProcessor* class. The operation computes the return value according to a simple algorithm, expressed in the *U2 Action Language*. Finally, *RequestHandler* sends *Reply* signal with the calculated value to the environment and destroys itself by performing a stop action.

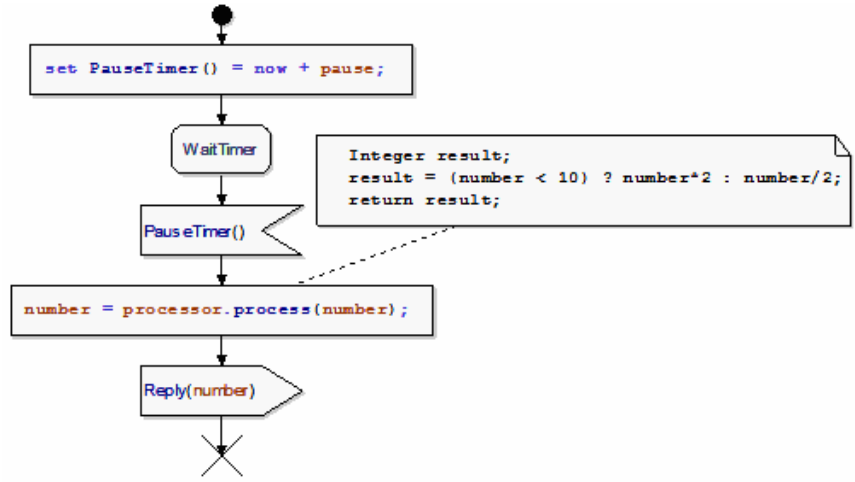

Figure 5.7. State machine of *RequestHandler* class

In order to run the complete model of the system, a build artefact needs to be created, compiled, and launched. An exemplary interaction between objects of the Counting Server is illustrated in a sequence diagram in **Figure 5.8**. It shows behaviour of the system in response for sending it a signal *Count(8, 5.0)*, where 8 is a number that is processed by the server, and 5.0 is the delay after which a *RequestHandler* starts computation of the result. As expected, the value returned by the system is 16.

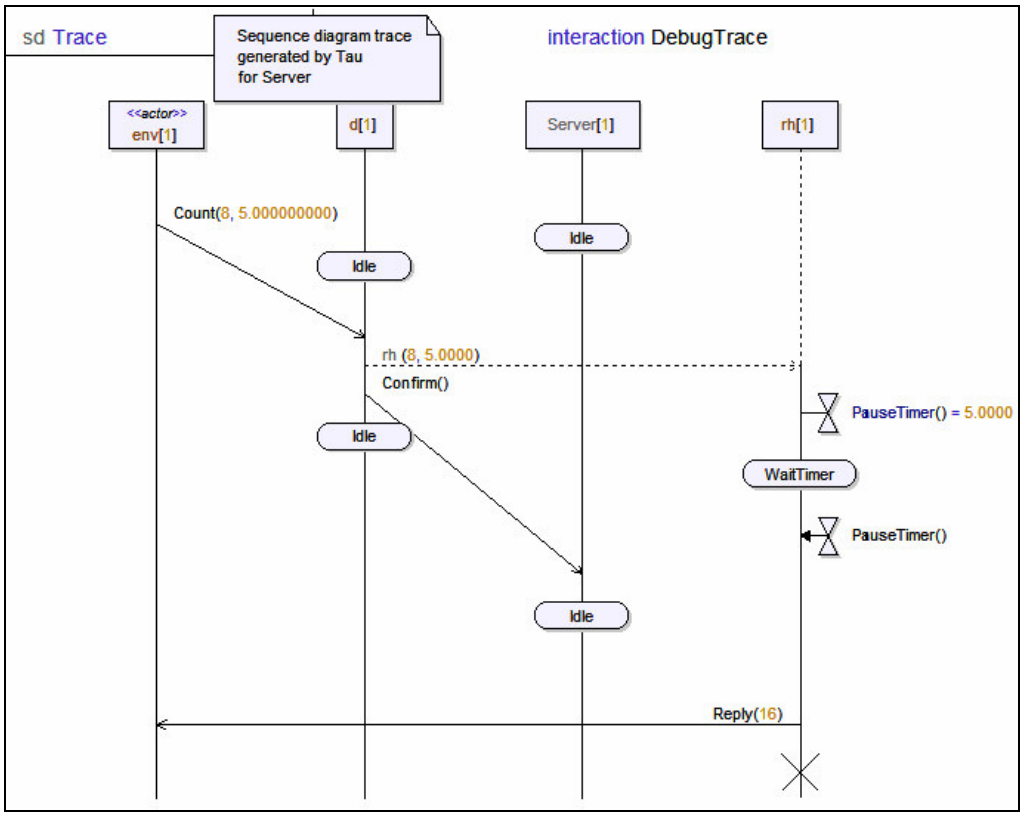

Figure 5.8. An exemplary interaction within the Counting Server

5.4 TAU's Compliance with UML 2.0

In the article on the future of software modelling, Kobryn [2004] foresees production and release of UML 2.0 tools supporting a wide variety of dialects loosely based on subsets of the UML 2.0 specification. These are subsets, because UML 2.0 is too large to be completely implemented by any vendor in one product release. The loose connection between the dialects and the UML 2.0 specification results mainly from the lack of a reference implementation and an appropriate test suite that could enforce and reliably measure the compliance.

A dialect of UML used in TAU is based on one of the submissions of the UML 2.0 Superstructure specification. However, in some cases TAU's UML differs even from this working version of the standard. This is mainly due to tool optimizations and foundation of some design decisions on yet earlier submissions. TAU also includes some extensions to the language, e.g. the possibility to use a textual syntax (the so-called U2 Textual Syntax) in conjunction with the graphical notation defined for UML. This syntax is based on C++ and Java and it additionally covers non-programming language concepts like stereotypes and tagged values.

The model repository of TAU is based on the so-called Object Model that is composed of around 200 metaclasses. Views of the underlying repository are provided by metamodels. An example of such a metamodel is the built-in TTDMetamodel which includes only the classes that are useful to be stereotyped, and omits almost all of the associations and attributes found in the core repository.

5.4.1 TAU Object Model

The metamodel[18] of TAU consists of eight inter-dependent packages (see **Figure 5.9**):

1. U2Build – contains one metaclass *Artifact* for modelling how to build and deploy a system;
2. U2Dynamic – contains entities for modelling dynamic behaviour of a system;
3. U2Entity – contains one metaclass *Entity* being the top-most superclass of all other metaclasses;
4. U2Persistence – contains entities for modelling how to store a model in a persistent way;
5. U2PredefinedTypes – contains the following primitive types: *einteger*, *eboolean*, *estring*, *evoid*, *ereal*, and *echaracter*;
6. U2Presentation – contains entities for modelling how to visualise a model using items like diagrams, symbols, lines, and text labels;
7. U2Scope – contains entities for modelling different kinds of scopes;
8. U2Static – contains some core entities for modelling static structural aspects of a system.

[18] Starting from this section, TAU metamodel refers to TAU Object Model.

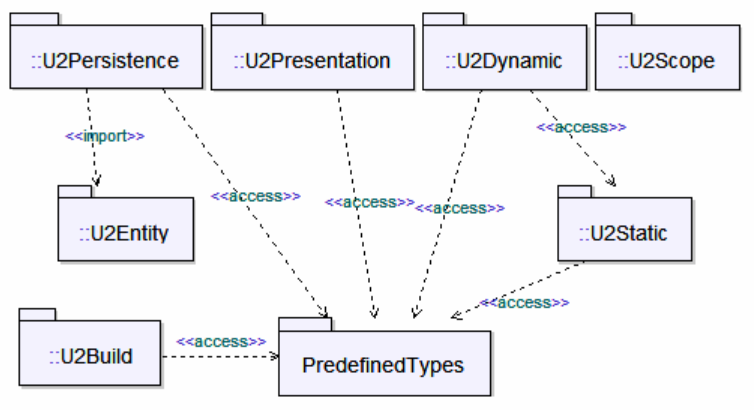

Figure 5.9. Packages of TAU Object Model

A detailed presentation and a description of the whole TAU Object Model or even an overview of the key aspects of the part that can be used to build executable UML models goes far beyond the scope of this thesis. However, without the knowledge of some of its fragments, it would be very hard, if not impossible, to comprehend in depth the details of the refactorings specifications from their catalogue (see Section 7). Therefore, a brief presentation of the parts of TAU Object Model that are particularly important in the context of these refactorings[19] is provided in Section 6.9. Moreover, a set of diagrams giving an overview of the whole TAU Object Model [Telelogic 2005] can be found on a CD attached to the thesis.

5.5 Communication between Classes

The structural part of a TAU executable model can be described with the use of collaborating active and passive classes. This fact is illustrated in **Figure 5.10**, which shows *TopLevelClass* having several parts – among others *ActiveClass1* and *ActiveClass2* – which may in turn be composed of other parts. All these active classes form the *architecture* of the system. However, in order to fulfil their responsibilities, active classes may use passive ones.

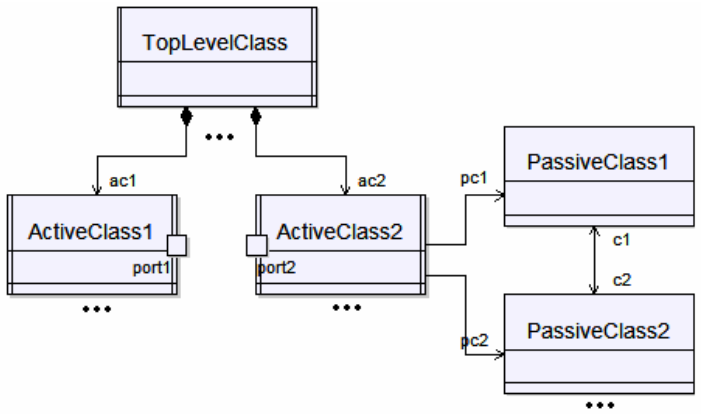

Figure 5.10. Scheme of the structure of a TAU executable model

[19] It is noteworthy that this part of TAU Object Model is not necessarily the most important in the context of another refactorings, e.g. refactorings specific for executable UML models.

Classes may communicate with each other by calling operations and sending signals. **Table 5.1** contains information on allowed methods of communication between two types of classes.

Table 5.1. Communication between classes in TAU

		FROM			
		PASSIVE		ACTIVE	
		Operation	Signal	Operation	Signal
TO	PASSIVE	YES	NO	YES	NO
	ACTIVE	NO	NO	YES	YES

From passive to passive

Passive classes may communicate with each other only with the use of operations invoked via associations. It is noteworthy that passive classes may not realize any interfaces (neither via realization nor via ports). Therefore, situations like the ones in **Figure 5.11** are illegal.

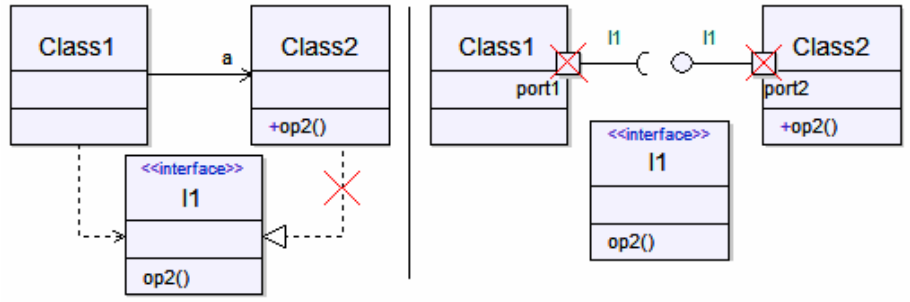

Figure 5.11. Passive classes may not realize any interfaces

From passive to active

A passive class may not initiate communication with an active one. It results from the constraints valid for TAU executable models stating that (1) methods of a passive class cannot call methods of an active one, and (2) passive classes cannot send signals. As passive classes do not have lifecycles described by state machines, the first rule prevents from any operational communication from them to active classes. Moreover, the second rule forbids passive classes to send signals to active ones. This constraint is not compliant with the basic causality model of UML 2.0 (see Section 4.2.2), which enables communication from passive classes to active ones via signals. Nevertheless, the only legal way to pass information from a passive class to an active one is the use of return parameters of operations.

From active to passive

The only way an active class may initiate communication with a passive one is the use of an operation call via an association. Passive classes cannot receive signals, mainly because they do not have state machines that would handle them.

From active to active

Active classes may communicate with each other with the use of both operations and signals. These are handled by their state machines as soon as an adequate state is reached, i.e. a state in which the particular operation call or signal receipt may trigger a transition to another state.

Judging from the exemplary projects supplied with TAU, the most common way of communication between active classes being parts is sending signals via connectors. This seems to result from the fact that sending a signal, as opposed to invoking an operation, is asynchronous and thus it does not suspend the execution of the state machine of the sender. Connectors may be used also as a communication medium for operation calls.

A top-level class (or a container) may use its composite associations to invoke operations or send signals to its parts. On the contrary, communication in the opposite direction, i.e. from a part to its owner is not possible, even if the composition association between them is navigable in both directions. Therefore, a container class may have behaviour ports that enable communication between its state machine (if it exists) and its parts.

Communication with the use of associations between parts – but only calling operations – is also technically possible. However, it is not justified, because active classes are supposed to be as much independent as possible and communicate via ports and connectors.

It is noteworthy that explicit connectors between ports are necessary only when there is an ambiguity in how signals can be transmitted in a model. An example of implicit communication is shown in **Figure 5.12**, where *Class1* and *Class2* are parts of the top-level active class called *RootClass*. *Class1* requires interface *I1* containing operation *op2()*, and *Class2* realizes it. Remarkable is the fact, that part *c1* may call *op2()* via *port1*, and – assuming that the state machine of *c2* is in a proper state – *op2()* in *c2* is invoked. This is possible, because in this model there is no uncertainty in signal communication.

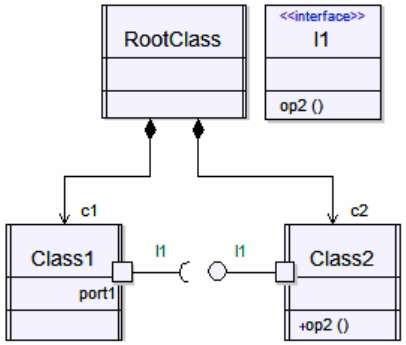

Figure 5.12. Implicit signal communication

6 EXECUTABLE UML MODEL REFACTORING

The main goal of this chapter is to present the results of an initial study on refactoring executable UML models. It is structured as follows: Section 6.1 discusses the role of behaviour in model refactoring; Section 6.2 emphasises the need for refactoring executable UML models; Section 6.3 investigates whether an executable subset of UML 2.0 could be used as a basis for a repository of a programming language-independent refactoring tool; Section 6.4 illustrates the phenomenon of change propagation in UML models; Section 6.5 defines *executable UML* in the context of the thesis; Section 6.6 argues that all Fowler's code refactorings [Fowler *et al.*1999] can be applied to both UML 2.0 as well as TAU models; Section 6.7 identifies six refactoring areas in TAU executable models; Section 6.8 shows how exemplary refactorings can be applied in the identified areas; Section 6.9 presents several fragments of TAU Object Model, which are particularly important in the context of refactorings triggered in ESPC area. Finally, Section 6.10 proposes a specification template for refactorings of executable UML models.

6.1 Introduction

In order to comprehend the role of behaviour in UML model refactoring, we differentiate two disjoint categories of *structural UML models* and *executable UML models*. The former ones specify only static structure of systems, and the latter ones can additionally specify their dynamic behaviour. As "there is no disembodied behaviour in UML" [OMG 2004], we assume that each model that has a behavioural part has a structural one as well, but the vice-versa does not have to be true.

The majority of previous studies presented in Section 3 focused on refactoring of structural models with the use of transformations – let us call them *structural refactorings* – derived from the majority of Fowler's structure-triggered refactorings. As well pre- and postconditions as transformation steps of these refactorings are simplified when compared to their code equivalents. This is caused by the lack of behavioural specification that normally would be taken into consideration in preconditions, and next transformed by a refactoring.

To answer a question whether these transformations are suitable for different kinds of UML models, we distinguish three states in which UML models can be during their development:

1. Structure of the system is modelled, but its behaviour is not yet defined.
2. Structure of the system is modelled, but its behaviour is defined only in the underlying code.
3. Both structure and behaviour of the system is modelled.

The first category encompasses both (a) structural models created in the context of simple forward engineering and (b) early executable models. These models can be safely refactored with the use of transformations originating from the majority of Fowler's structure-triggered refactorings. However, the question that arises in the context of this model category is whether these transformations are still refactorings. As argued by Fowler [2004], one cannot refactor anything that does not have a well-defined behaviour. Otherwise, how can one guarantee the behaviour-preservation of these transformations? This argument speaks for calling application of refactoring transformations to structural and early executable models – *restructurings.* On the

other hand, can a transformation not preserve properties of something that does not yet exist?

The second category covers structural models created with the use of round-trip engineering. The models in this state outwardly do not differ from the ones from the first group. However, application of above-mentioned simplified refactorings to these models and a successive synchronization of the underlying source code with an altered model will likely introduce errors. Therefore, suitable versions of refactorings for this category of models need to somehow (1) obtain information necessary for their preconditions from the corresponding source code, and (2) update not only structural but also behavioural parts of this code to reflect changes introduced to the model. The refactoring of UML models from this category with the emphasis on the problem of vertical software consistency is considered in more detail in Sections 3.5 and 3.6.

The third category encompasses late executable models. By "late" understood is the presence of a behavioural part of a model that will eventually allow execution of the modelled application by e.g. automated source code generation. Structural refactorings applied to a model from this category transform only its structural part, what usually causes horizontal inconsistency between its structural and behavioural descriptions. Appropriate versions of refactorings for this category of UML models should take into account in their preconditions and update both structural and behavioural information available in the model. More concretely, it is necessary to capture at least the actions that create and delete objects, get and set attribute values, and call operations [Kazato *et al*. 2004].

6.2 Motivation for Refactoring Executable UML Models

In this section, we present two exemplary common scenarios emphasising the need for refactoring of executable UML models.

In the first scenario, an inexperienced modeller builds a too simple class model, what in turn enforces construction of too large and too complex state machines describing the lifecycles of the initially identified classes. Mellor & Balcer [2002], in their book on Executable UML [2002], illustrate this problem of "incomplete factoring of classes" on an example[20], in which a part of the responsibility of a class *Order* is extracted into a new class *ShoppingCart*. This transformation, called by the authors *Refactoring Behaviour*, leads to the simplification of as well structure of *Order* as its statechart diagram. In this case, the refactoring is triggered on the class model, and the state machines are supposed to be simplified in a behaviour-preserving way, i.e. the externally observable behaviour of the system cannot change. An attempt to carry out this task in an *ad hoc* manner, i.e. without the use of behaviour-preserving refactorings, would likely lead to a failure.

In the second scenario, an inexperienced modeller fails to construct good state machines for previously properly identified classes. Mellor & Balcer [2002] provide an example[21] of two statechart diagrams representing the lifecycle of a *Shipment* class.

[20] Chapter 12.2 "Reworking the Class Diagram".
[21] Chapter 12.1 "Statechart Diagram Construction Techniques".

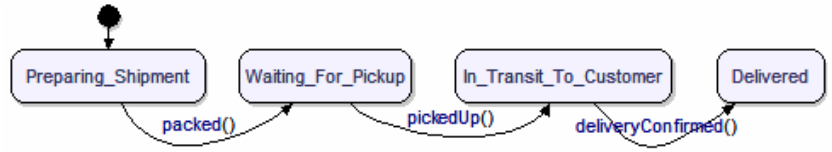

Figure 6.1. A well-shaped state machine of *Shipment* class [Mellor & Balcer 2002]

The first statechart is built in accordance with "Modelling Intention" technique, in which a class relies on itself in fulfilling its lifecycle (see **Figure 6.1**), and the second one is constructed with the use of an opposite approach that leads to a "spider" shape, with a central state that waits for requests and a set of "legs" that respond to each of them (see **Figure 6.2**).

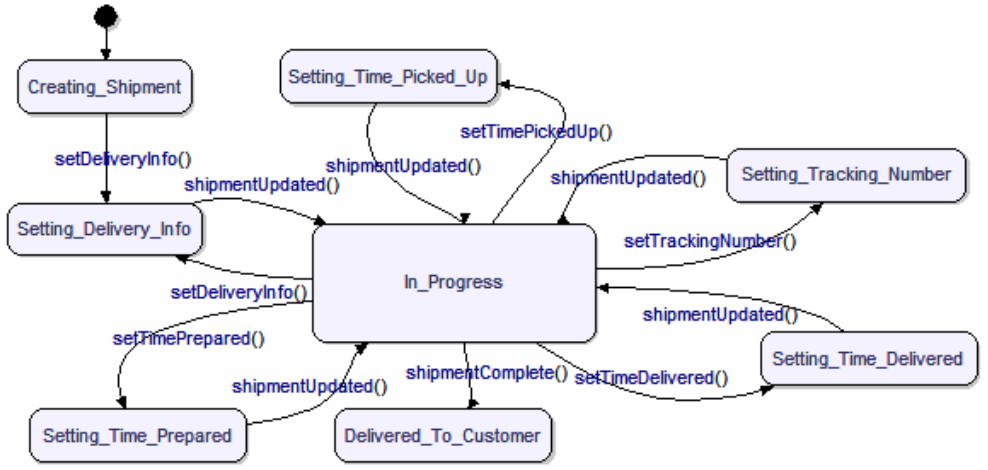

Figure 6.2. A "spider-shaped" state machine of *Shipment* class [Mellor & Balcer 2002]

From an external viewpoint, both statecharts may describe the same behaviour of the Shipment class. However, from the development perspective, the first statechart is more readable, comprehensible, and results in a much more maintainable model than in the case of the "spider-shaped" one, which obscures key sequencing issues, what causes that e.g. it can be tricky to determine the state in which an object really is. If for some reason, a modeller ends up with "spider-shaped" statecharts and subsequently with an unmanageable model, the only alternative for removing and creating the part of the model from scratch is to refactor these state machines, consequently modifying also the class model. The use of this technique enables the modeller to obtain, in a safe and stepwise manner, a well-formed model from an ill-formed one.

These two scenarios are not the only ones that motivate refactoring of executable models. Moreover, even very experienced modellers can benefit from using this technique. Mellor & Balcer [2002] emphasise that the process of building Executable UML models is incremental and iterative – starting from simple models with limited capabilities, additional functionality is added incrementally. As one cannot foresee all the emerging requirements, refactoring can be used to make the model more maintainable, so that the new functionality can be next incorporated to it faster than to a bad-factored one.

6.3 Language-Independent Code Refactoring with UML

Starting from Section 6.5, executable UML is treated as a concrete language used for development of models that are expressive enough to be compiled to source code in any programming language. This fact indicates that the opposite process, i.e. regeneration of a model from a code, is also likely to be possible, what gives rise to the question whether an executable subset of UML 2.0 could be used as a basis for a repository of a programming language-independent[22] refactoring tool? Such a tool would be very useful for companies that for some reasons hesitate to shift from the traditional to the model-driven development, in which it would be redundant, but still want to benefit from round-trip engineering. Van Gorp *et al.* [2003a] describe an example of this kind of a tool, which however bases not on UML 2.0 but the on *GrammyUML* metamodel.

To answer the above-posed question one has to first identify the information needed to reason about code refactoring, and next determine whether this information can be expressed with the use of UML 2.0. Particularly important are constructs that may appear in method bodies, because UML actions that may be used to model them have been rapidly evolving over successive versions of the UML specification.

According to Van Gorp *et al.* [2003a; 2003b], a stable basis to reason about consistent refactoring is the notion of access-, call-, and update-behaviour augmented with the concept of type checking and type casting. It turns out that the UML 2.0 Actions package contains metaclasses to model all these constructs.

Attributes of classes can be accesses with the use of *ReadStructuralFeatureAction*, and updated by as well *ClearStructuralFeatureAction* as two subclasses of *WriteStructuralFeatureAction*, namely *AddStructuralFeatureValueAction* and *RemoveStructuralFeatureValueAction* (see **Figure 6.3**).

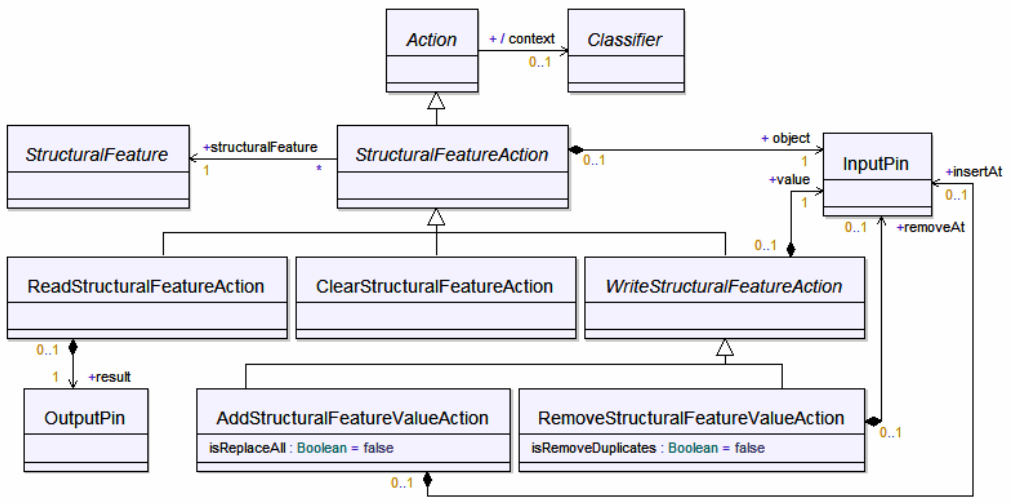

Figure 6.3. Structural Feature Actions [OMG 2004]

CallOperationAction is suitable for modelling operation calls (see **Figure 6.4**).

[22] Since UML is object-oriented, concerned here is the family of object-oriented languages.

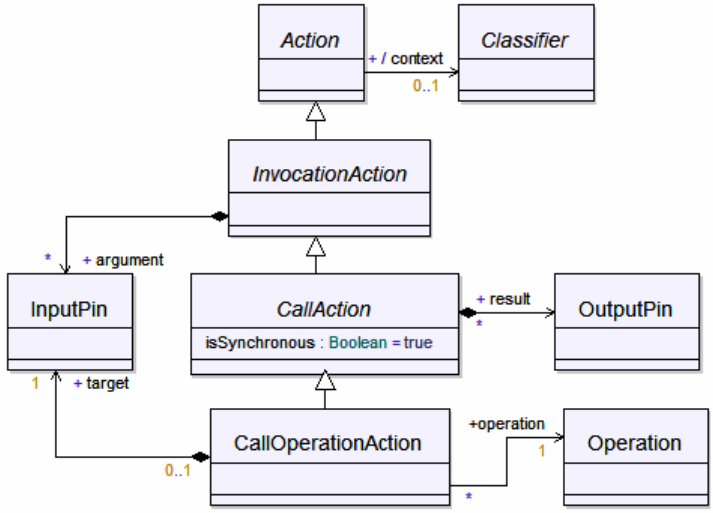

Figure 6.4. *CallOperationAction* [OMG 2004]

The notion of type checking and type casting is expressible by *ReadIsClassifiedObjectAction* (or *ReadExtentAction*) and *ReclassifyObjectAction* respectively (see **Figure 6.5**).

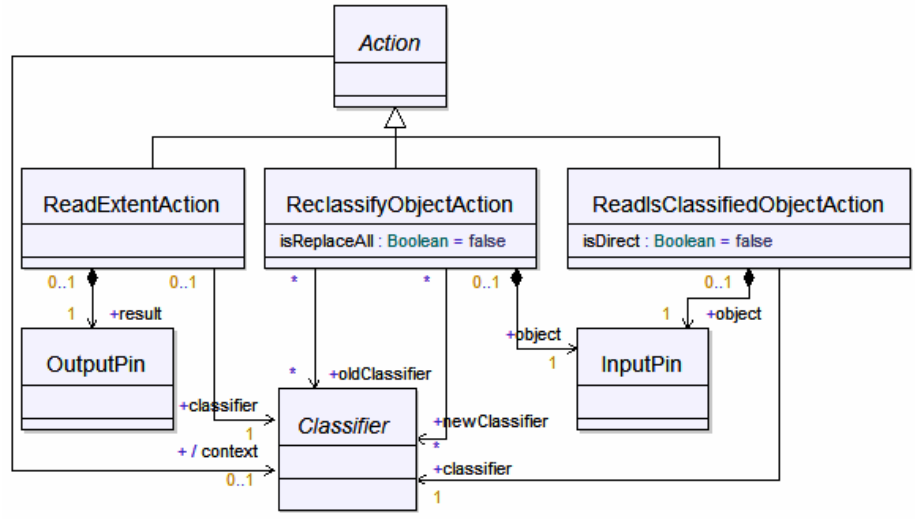

Figure 6.5. Object Actions (part) [OMG 2004]

As stated by Mens & Tourwé [2004], a tool or a formal model for refactoring, besides being sufficiently abstract to be applicable to different programming languages, should also provide the necessary hooks to add language-specific behaviour. In the case when the set of actions offered by UML 2.0 is not sufficient to model them, one can use *OpaqueAction* introduced to define actions with implementation-specific semantics.

6.4 Change Propagation in UML Models

As already stated in Section 3.1, according to UML specification [OMG 2004] diagrams are not parts of UML models, but just graphical representations of their parts. However, since diagrams are expected to be stored together with models is a repository, some state-of-the-art UML CASE tools like Telelogic TAU extend UML

metamodel and treat diagrams and their contents as e.g. instances of subclasses of *PresentationElement*[23] metaclass, being a subclass of *Element* – the root UML metaclass. Disregarding the way the diagrams are stored by tools, we distinguish two kinds of modifications – *model modifications*, and *diagram modifications*, where each former one may trigger zero to many latter ones. This distinction is based on an assumption that diagrams are automatically regenerated from a model repository after each change of any model element, so that they always reflect an up-to-date state of a part of the model they show.

In some cases, a modification of a model element in one part of the model seems to causes several other model modifications in other parts of this model. For example, if the name of an attribute *attrA* is changed to *attrB*, then this modification is reflected in all parts of the model, in which this attribute is referenced, i.e. each occurrence of attribute *attrA* in bodies of all operations[24] will be substituted with *attrB* (see **Figure 6.6**).

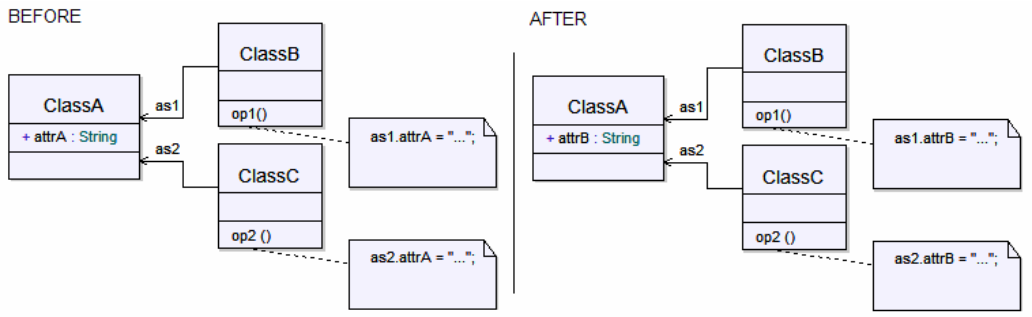

Figure 6.6. Change propagation after renaming an attribute

However, these automatic updates of operations' bodies are ostensible, and the only actual model modification that occurred in this example was the change of the attribute's name. The explanation of this phenomenon requires comprehension of the UML metamodel, on which – at least conceptually – are based model repositories of all UML metamodel-driven tools.

In the example, each instance of *ReadStructuralFeatureAction* metaclass that references attribute *attrA* has an association, represented by *structuralFeature* attribute, to the same instance of *Property* (subclass of *StrcturalFeature*) metaclass being attribute *attrA*. This *Property* has a metaattribute *name* (inherited from *NamedElement*), and after the change to the value of this metaattribute, associations represented by *structuralFeature* attributes still point at former *attrA*, which is now called *attrB*. The corresponding part of the UML 2.0 metamodel is shown in **Figure 6.7**.

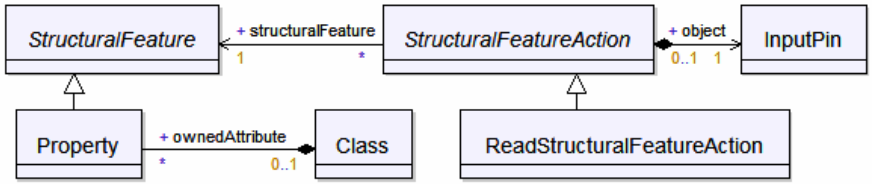

Figure 6.7. *ReadStructuralFeatureAction* [OMG 2004]

[23] *PresentationElement* is not a UML 2.0 metaclass
[24] Generally – in all *Activities*

In the context of refactoring, Sunyé *et al.* [2001; 2002] noticed this observable fact that some model elements shown in one view, e.g. in class diagrams, may have a direct connection to the elements of other views, and therefore some refactorings that apply to elements in one view may have an impact on different UML views. This phenomenon is often called *change propagation*. In some cases, a change propagates in a way that no additional modifications are needed to restore the consistency and correctness of a model. In this simple example, this fact was very helpful, because it prevented us from the necessity of manual adjustment of attribute names in operation bodies. However, the lack of precise knowledge of UML metamodel could lead to misguided conclusions that e.g. in **Figure 6.8**, after moving attribute *attrX* from class *ClassB* to *ClassC*, operation body of operation *op1()* from *ClassA* would contain the following expression: *Integer i = **asC**.attrX*.

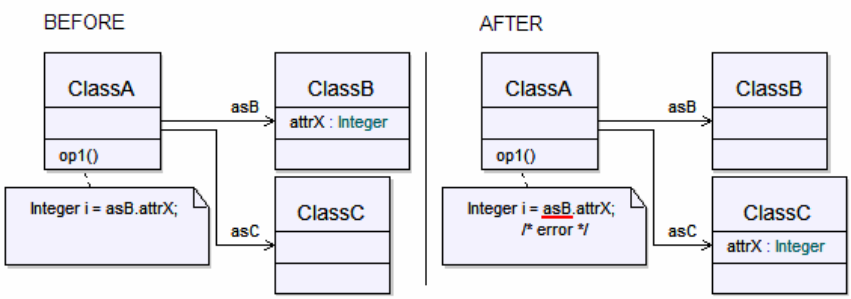

Figure 6.8. Change propagation after moving an attribute

However, each instance of *StructuralFeatureAction* metaclass obtains the object, whose structural feature is to be read or written, not from the value of the *qualifiedName* metaattribute of this structural feature, but via the *object* association (see **Figure 6.7**).

As it will be shown in next sections, even simple rename refactorings, i.e. Rename Attribute, Rename Operation, and Rename Class, cannot be fully accomplished by application of only corresponding rename operations – even they may require additional adjustments.

6.5 Executable UML in the Context of the Thesis

Although the UML 2.0 Superstructure specification [OMG 2004] defines the run-time semantics of the executable subset of UML, an attempt to refactor executable models fully compliant with the standard has been abandoned due to the following reasons:

1. The specification has been rapidly evolving[25], and its currently available version is not yet the final one;
2. The executable subset:
    a. contains many semantic variation points,
    b. contains overlapping constructs,
    c. is not streamlined strictly enough.

Therefore, it has been decided to focus on refactoring of models that can be built and executed in one of the state-of-the-art UML CASE tools that fulfil two following criterions:

[25] See the list of almost 900 issues submitted to UML 2.0 Superstructure Finalization Task Force at http://www.omg.org/issues/uml2-superstructure-ftf.html.

1. Models executed by this tool have to be as much compliant as possible with the UML 2.0 standard [ibid.];
2. The tool has to be either free-of-charge or available at no less than one university at which this thesis is written.

The only tool that satisfies both requirements is Telelogic TAU Generation2. For that reason, the catalogue of model refactorings in Section 7 concerns executable UML models that can be compiled to TAU Model Verifiers, i.e. applications instrumented to support simulation, tracing, and detailed debugging at the UML level (refer to Section 5). This category of models has been chosen, mainly because (1) they are proven to be executable, and (2) their behaviour can be visualised and thus easily verified.

6.6 Determination of Candidate Refactorings

The work on composition of the catalogue of executable UML model refactorings starts with determination of *candidate refactorings*, i.e. Fowler's refactorings, which can be basis for model ones. In other words, chosen are going to be only these refactorings from Fowler's catalogue, from which executable UML model refactorings can be easily derived.

The four refactorings from Fowler's *Big Refactorings* group are not taken into consideration, since they seem to be too ambiguous to become simple model transformations. Instead, they are rather examples how different refactorings can be employed in order to perform a serious architectural transformation of a piece of software. Therefore, model transformations derived from Tease Apart Inheritance, Extract Hierarchy, and Separate Domain from Presentation[26] could be performed with the use of a sufficiently expressive set of model refactorings. The refactoring Convert Procedural Design to Objects is hardly expressible in UML, because it concerns systems created with the use of structural, as opposed to object-oriented, paradigm.

The selection is based on an obvious assumption that only the refactorings that are triggered on elements existing in a language may by applied to programs written in this language. For example, it is not possible to perform Replace Nested Conditional with Guard Clauses in a language that does not support nesting of conditional statements. Therefore, a mapping between all twenty-one code trigger-elements and their counterparts in TAU (in the form of TAU Object Model metaclasses) has been prepared (see Appendix F). Additionally, a similar mapping is provided between these metaclasses and their equivalents in the UML 2.0 metamodel [OMG 2004], what is a great starting point for a future work on refactoring of executable UML models fully compliant with the standard. Moreover, this comparison shows how incompatible with the UML 2.0 metamodel is the one implemented in TAU.

In the effect of the work on the mapping, equivalents for all code trigger-elements have been identified – both among TAU and among UML 2.0 metaclasses. This implies that all Fowler's code refactorings can be triggered on as well TAU as UML 2.0 executable models. Assuming that the solution domains of refactorings do not require any additional language elements or constructs, all of them may be performed on executable UML models.

[26] Assuming the existence of a GUI modelling library.

6.7 Refactoring Areas in TAU Executable Models

As already indicated in Section 2.7.1, each code refactoring may be triggered on either a structural (e.g. field) or a behavioural (e.g. temporary variable) element. This implies that in programs written in object-oriented programming languages there are two *refactoring areas*, namely (1) structure – inter-related classes and their features, and (2) behaviour – bodies of methods.

In the context of UML, a refactoring area may be defined as a certain part of a model containing particular trigger-elements. Refactorings applicable for UML 1.x models are only the structure-triggered ones, because in these models there is only one refactoring area – the structure. On the other hand, in TAU executable models, six refactoring areas can be distinguished (see **Figure 6.9**).

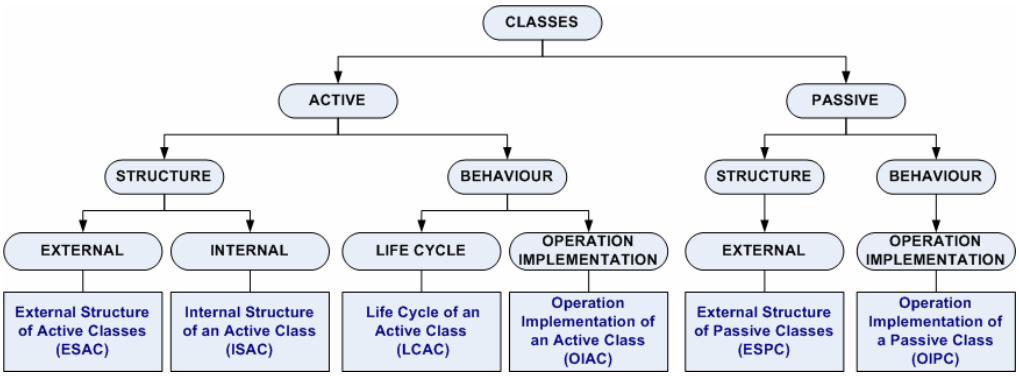

Figure 6.9. Six refactoring areas in TAU executable models

6.7.1 External Structure of Active Classes (ESAC)

The area consists mainly of active classes that have attributes, operations, and ports that require and realize single signals and/or whole interfaces. Additionally, active classes may have (composite/shared) associations to passive classes and composite associations to their parts being other active classes. All these elements may be show in class diagrams, and some of them (e.g. parts and ports) additionally in composite structure diagrams. Candidates for transformations that can be triggered on model elements from this area are all of the structure-triggered code refactorings. However, in many cases their practical realization may considerably differ from the ones provided by Fowler [Fowler *et al.* 1999]. Moreover, in the area there are potentially many other, so far unidentified refactorings triggered on among others ports, signals, timers, and interfaces.

6.7.2 Internal Structure of an Active Class (ISAC)

The area consists of active classes that are parts of their container, and which communicate with each other by sending signals and invoking operations via ports wired by connectors. Some aspects of this area may be illustrated in class diagrams, but e.g. connectors – only in composite structure diagrams. As composite structures are new to UML 2.0, so far, there exists no literature concerning refactorings applicable to this area. Nevertheless, these transformations would deal mainly with reorganization of internal structure and communication infrastructure of active classes, and thus they have no equivalents among code refactorings.

6.7.3 Life Cycle of an Active Class (LCAC)

The area constitutes implementation of a default state machine of an active class, represented in a statechart diagram. In this area, there are two kinds of trigger-elements: (1) states and transitions between them, and (2) (elements of) compound actions on transitions. Refactorings triggered on the elements from the first group are mainly specific versions of some behaviour-triggered code refactorings. Other, but already UML specific, transformations identified and described by Sunyé *et al.* [2001] and Boger *et al.* [2003] can be applied to the elements from the second group.

6.7.4 Operation Implementation of an Active Class (OIAC)

The area constitutes implementation of an operation belonging to an active class. In the context of active classes, this implementation may be either state or stateless. However, the former solution introduces only new presentation elements for corresponding triggers being the same model elements, namely (elements of) various actions, as in the latter case. Refactorings triggered on elements from this area may be derived from code behaviour-triggered ones. However, their realizations may differ from the ones provided by Fowler [Fowler *et al.* 1999] due to the possibility of among others communication with the use of signals and via connectors.

6.7.5 External Structure of Passive Classes (ESPC)

The area consists of passive classes that may have attributes and operations as well as (composite/shared) associations and generalizations to other passive ones. All these elements may be show in class diagrams. Candidates for transformations that can be triggered on model elements from this area are the same as in the case of ESAC, i.e. all of the structure-triggered code refactorings. However, their practical realizations are usually simplified with respect to their equivalents from ESAC.

6.7.6 Operation Implementation of a Passive Class (OIPC)

The area constitutes implementation of an operation belonging to a passive class. In the context of passive classes, this implementation may be only stateless, i.e. in the form of actions written in *U2 Action Language* contained in a text diagram. As in the case of OIAC, the refactorings triggered on elements from this area may be derived from code behaviour-triggered ones, but their practical realizations are usually simplified with respect to their equivalents from OIAC.

6.8 Application of Exemplary Refactorings

To facilitate the understanding of refactoring areas, we choose four transformations[27] and show how they can be applied to an exemplary TAU executable UML model of a satellite[28]. The selected transformations are:

1. Area ISAC – Extract Port[29] – triggered on a port of an active class, which is used for communication with several different parts. It relies on creating

[27] It is worthy noting that refactorings Replace Method with Method Object and Hide Delegate can be triggered also on the elements in the OIPC and ESAC areas, respectively.

[28] For the sake of conciseness, only these parts of the model, which are important in the context of particular transformations, are presented.

a new port and reconnecting some connectors of the old one to the new one.

2. Area LCAC – Group States – triggered on a simple state in a default state machine of an active class. It relies on transforming the state into a composite one, and thus reduces the number of redundant transitions.
3. Area OIAC – Replace Method with Method Object – triggered on an implementation of an operation of an active class, which is too long and cannot be decomposed with the use of other refactorings. It relies on turning the operation into a class.
4. Area ESPC – Hide Delegate – triggered on a passive class. It relies on encapsulating it from other ones, and thus reduces the coupling between classes in the model.

6.8.1 Area ISAC – Extract Port

The top-level active class of the model is *Satellite*, which has several parts typed by *EarthCommunicator*, *Navigator*, and *InstrumentsController*. As can be observed in **Figure 6.10**, *pOutput* port of *earthCommunicator* serves for communication with two different parts with the use of two semantically unrelated signals – *plan*, containing the most recent plan of the mission, and *command*, carrying new instructions for scientific and navigational instruments. A refactoring that should be triggered on *pOutput* is Extract Port.

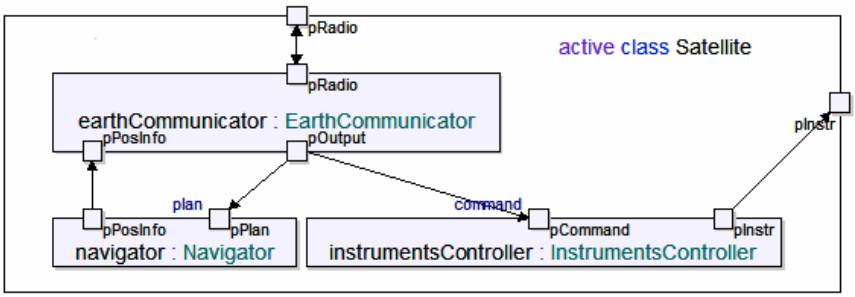

Figure 6.10. A simplified composite structure diagram of *Satellite*

As the result of the refactoring, a new port *pCommand* is added to *EarthCommunicator*, and the connector that transmits *command* is reconnected to it. Next, on a class diagram showing *EarthCommunicator*, *command* is moved from to the list of signals required by *pOutput* to the one belonging to *pCommand*. Assuming that in all output actions – be it in a default state machine of *EarthCommunicator* or in bodies of its operations – signals are sent without *via* keyword, the refactoring finishes, otherwise each expression "*[output] command(instr) via pOutput*" has to be changed to "*[output] command(instr) via pCommand*". Subsequently, one can apply Rename Port refactoring to *pOutput* in order to give it a more meaningful name, e.g. *pPlan* (see **Figure 6.11**). Finally, one can consider merging *pPosInfo* and *pPlan* in both *EarthCommunicator* and *Navigator* with the use of Merge Ports refactoring.

[29] This refactoring has not been previously mentioned in the literature.

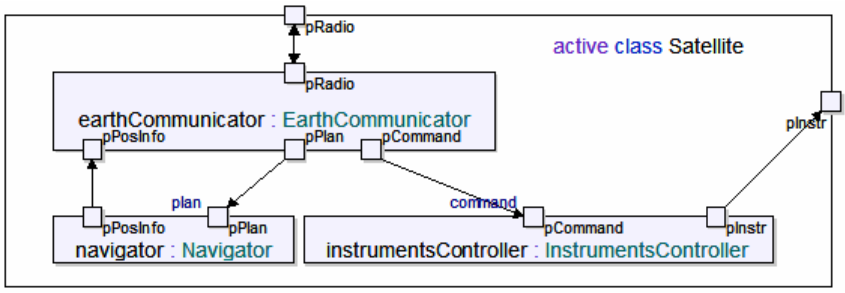

Figure 6.11. A simplified composite structure diagram of *Satellite* after triggering Extract Port and Rename Port on *pOutput* port

6.8.2 Area LCAC – Group States

The lifecycle of *InstrumentsController* is defined by a state machine shown in **Figure 6.12**. Just after creation, an instance of the class finds itself in *Idle* state, in which it awaits for *command* signal sent by *earthCommunicator*. The signal triggers a transition to *Decoding* state. Next, after going through *Calculating* and *Encoding* states, the state machine reaches *Adjusting* state, in which it adjusts every 10 ms the instruments, as long as new instructions appear.

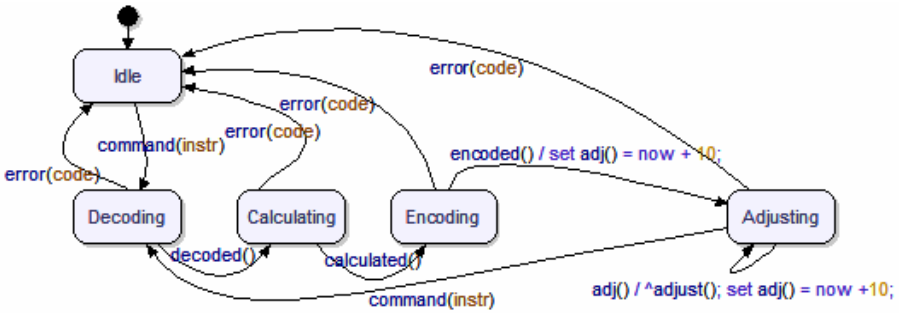

Figure 6.12. A state chart diagram showing implementation of a default state machine of *InstrumentsController*

It can be noted that from each state there is a transition to *Idle* state triggered by *error* signal. These redundant transitions can be eliminated by the application of a refactoring known as Group States [Sunyé *et al.* 2001]. During the refactoring, first a new state *Working* is created. Next, three transitions triggered by *error* from *Calculating*, *Encoding*, and *Adjusting* are deleted, and the one from *Decoding* is reconnected to the new state, as well as a transition triggered by *command* from *Idle*. Subsequently, a new state machine is created in *Working*, what makes this state composite. Finally, *Decoding*, *Calculating*, *Encoding*, and *Adjusting* are moved together with their transitions to the new state. The transformation is completed by addition of a start symbol. Its effects can be seen in **Figure 6.13**.

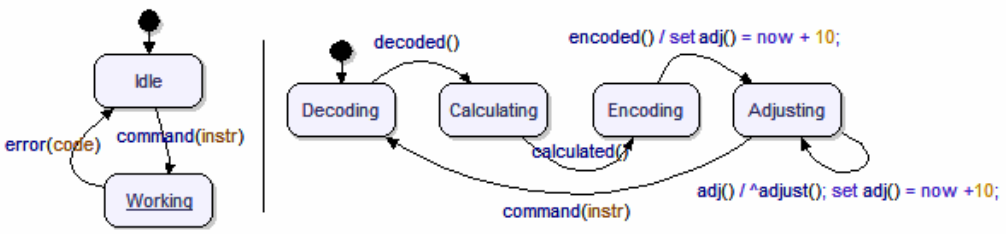

Figure 6.13. Two state chart diagrams showing implementation of a default state machine of *Instruments Controller* (left) and *Working* state (right), after applying Group States

6.8.3 Area OIAC – Replace Method with Method Object

Navigator component of *Satellite* is a compound object, which has as one of its parts an instance of an active class *CollisionDetector*. The class has among others an operation *avoid*, that takes as a parameter an instance of *Collision* class obtained from invocation of *detect* operation. The responsibility of *avoid* is to (1) determine how to avoid the collision and (2) return the result of computation in the form of an instance of *AvoidancePlan* class. The problem with *avoid* is that its operation body is too long, what is an unequivocal symptom of *Long Method* bad smell [Fowler *et al.* 1999]. However, the operation uses its local variables *dimX*, *dimY*, and *dimZ* in such a way that even after application of Replace Temp with Query, its decomposition with the use of Extract Method is impossible. Therefore, instead of Extract Method, Replace Method with Method Object is triggered on *avoid*. The described part of the model before the transformation is shown in **Figure 6.14**.

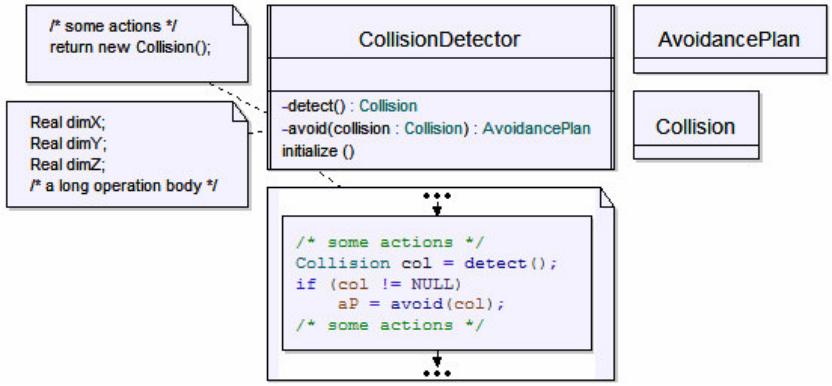

Figure 6.14. A class diagram showing a situation qualifying for application of Replace Method with Method Object

In the first step, a new passive class is created and named by the operation. Next, an attribute for each temporary variable (*dimX*, *dimY*, and *dimZ*) and the parameter (*collision*) of *avoid* is created in the new class. Then, *Avoid* is given a constructor that initializes *collision* attribute. Subsequently, in the new class a new operation *compute* is created with the body copied from *avoid*. Next, all temporary variables are removed from the body of *compute*, and the body of *avoid* is replaced with one that creates an instance of *Avoid* and calls *compute*. The effect of the transformation is shown in **Figure 6.15**. Because all the previous local variables of *avoid* are now attributes, one can easily decompose the operation with the use of Extract Method.

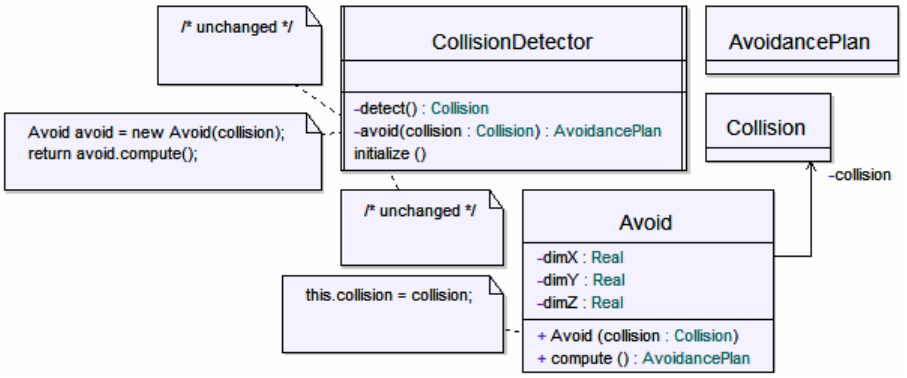

Figure 6.15. A class diagram showing *CollisionDetector* after application Replace Method with Method Object on the body of *avoid* operation

It is worth to observe that the model after Replace Method with Method Object will not work properly if *avoid* invokes any operation of *CollisionDetector* or any operation of any other class accessible from it. In such a situation, Fowler [Fowler *et al.* 1999] advises to give the new class an attribute for the object that hosts the original operation (the *source object*), initialize it in the constructor, and use it for any invocations of operations on the original class. However, in this case it cannot be done, because *CollisionDetector* is an active class, and passive classes are not allowed to invoke any operations of active ones. Moreover, *avoid* can include neither *output actions* responsible for sending signals nor actions concerning timers, i.e. *timer set* or *timer reset actions*, because these are also not permitted in operation bodies of passive classes.

6.8.4 Areas ESAC & ESPC – Hide Delegate

The refactoring discussed here relates to the Fowler's statement saying that "one of the keys, if not the key, to objects is encapsulation [Fowler *et al.* 1999]." In general, the less each class in a model needs to know about other classes, the less possible is that a change in one place causes the necessity to adjust other parts of the model, what makes the model maintenance easier and cheaper. For instance (see **Figure 6.16**), let us consider a situation in which a client class (*PlanSupplier*) invokes an operation (*getDestination*) defined on one of the attributes (*plans* accessible via *getDestination*) of a server class (*PlanStorage*).

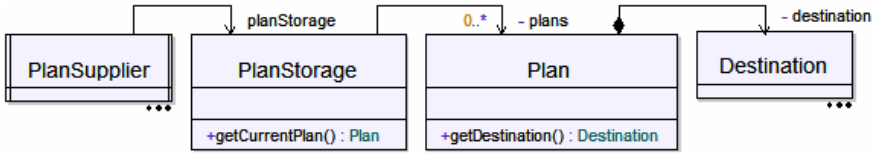

Figure 6.16. A class diagram showing a situation qualifying for application of Hide Delegate

As the client has to know about the delegate class (*Plan*), each change of the delegate may propagate to the client. This redundant dependency can be removed by placing a simple delegating operation on the server, which hides the delegate. A refactoring that performs this task is known as Hide Delegate. First, *getCurrentPlan* is renamed with the use of Rename Operation to *getCurrentDestination*. Then, assuming that the most recent plan is always the first one in the *plan* collection, the body of the operation is changed from "*return plan[0]*" to "*return plan[0].getDestination()*". Finally, each statement in the form of "*Destination d* =

planStorage.getCurrentDestination().getDestination()"[30] is replaced by "*Destination d = planStorage.getCurrentDestination()*". These statements may occur in bodies of all operations of both passive and active clients of *PlanStorage*, as well as on transitions in state machines of active ones. After the refactoring, changes become limited to the server and do not propagate to the client (see **Figure 6.17**).

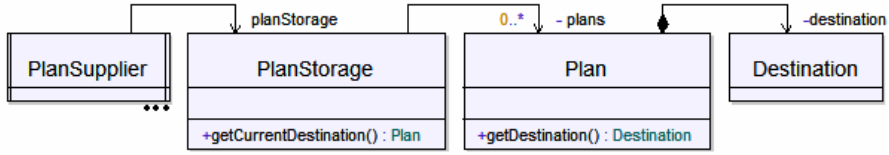

Figure 6.17. A class diagram showing the effect of application of Hide Delegate

6.9 Metamodel of ESPC Area

The initial catalogue of TAU refactorings in Section 7 contains specifications of transformations triggered mainly on elements from the ESPC (External Structure of Passive Classes) area, namely:

- Class
- Class::Attribute
- Class::Operation
- Class::Operation::Parameter

Comprehension of formal specifications of these refactorings requires detailed knowledge of the part of TAU metamodel that enables modelling of external structures of passive classes, and how constructs from this part are related to, i.e. use or are used by, elements from other views. Therefore, this Section presents shortly several fragments of TAU Object Model, which are particularly important in the context of refactorings triggered in ESPC area, organized around attributes and operations of passive classes.

6.9.1 Attribute of a Passive Class

An attribute of a class, modelled with the use of *Attribute* metaclass[31], is owned by a class via *ownedMember* association, and all attributes of a class can be determined using *attribute* association. The type of an attribute can be determined via *type* association inherited by *Attribute* from *Typed* metaclass, and a class in which it is defined – among others with the use of *namespace* association inherited by *Class* (an indirect superclass of *StructuredClassifier*) from *Namespace*, or *source* association inherited from *Signature* being also a superclass of *Class* (see **Figure 6.18**).

[30] It is noteworthy that the presence of *getCurrentDestination* in this statement is caused by application of Rename Operation.
[31] For the taxonomy of *Attribute* metaclass refer to Appendix G.

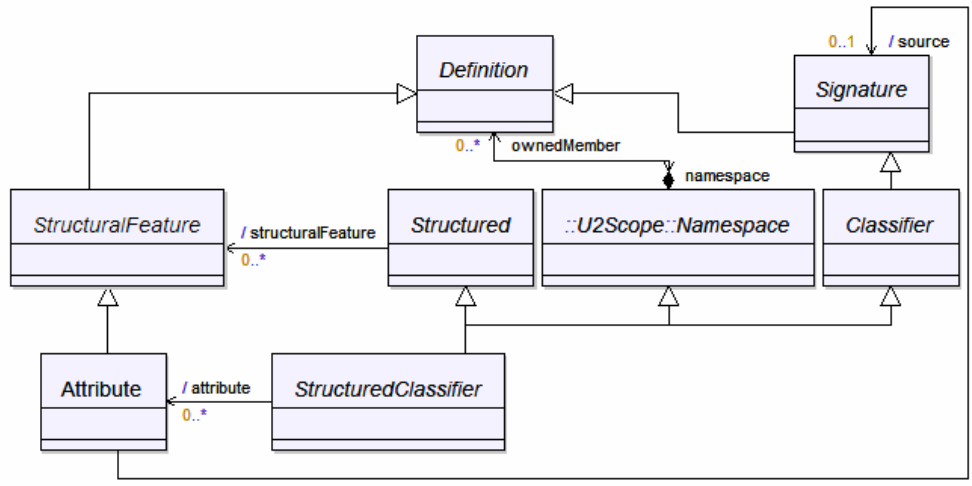

Figure 6.18. *Attribute* metaclass in the role of an attribute of a class

In the context of TAU executable UML, an association is a relationship between two signatures, indicating that instances of these signatures will be directly or indirectly connected to each other. An association has two association ends, represented as attributes accessible via *associationEnd* (see **Figure 6.19**).

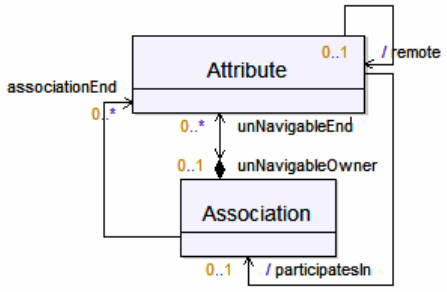

Figure 6.19. *Attribute* metaclass it the role of an association end

6.9.1.1 Attribute reference and access

The only way to reference/access an attribute of a class from a foreign one is to use an instance of *FieldExpr*, being one of direct subclasses of *Expression*, which can be owned by *ExpressionAction* via *expression* association (see **Figure 6.20**). *FieldExpr* points referenced/accessed attribute using *field* association. Additionally, it owns an *Expression*, being for example an identifier (*Ident* inherits from *Expression*) pointing another, certainly local, attribute via *definition* association.

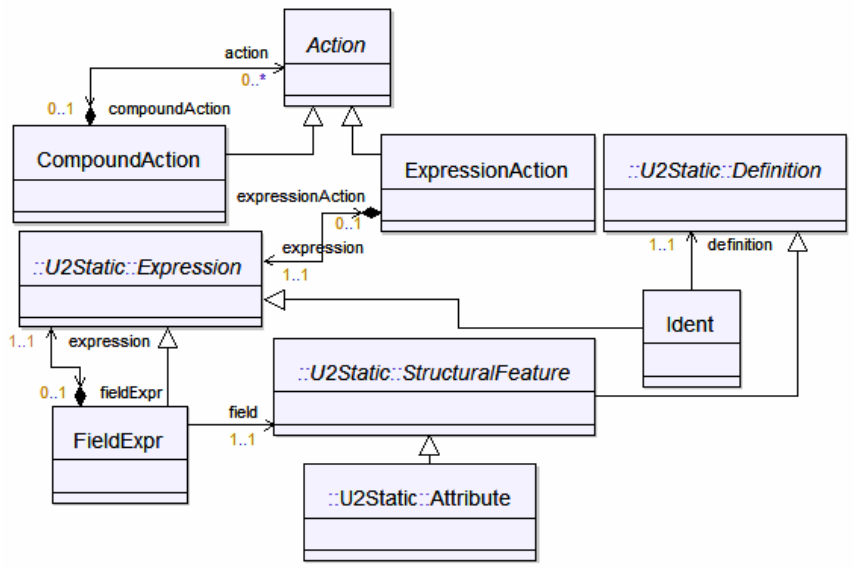

Figure 6.20. *Attribute* metaclass used by *FieldExpr* and *Ident* in actions

Locally, i.e. in the scope of a class owning it, an attribute can be read/wrote as a *definition* of an *Ident*, or alternatively as a *field* in a *FieldExpr*, but it the latter case, only if it is typed by a class in which it is defined.

Instances of *CompoundAction* from **Figure 6.20**, containing – probably nested – field expressions and identifiers pointing attributes of passive classes, can potentially model following elements:

- implementations of operations:
  - in passive classes,
  - in active classes,
- implementations of "initialize" state machines of active classes in:
  - actions performed on triggered transitions,
  - conditions evaluated in guards of guarded transitions.

6.9.2 Operation of a Passive Class

Operation[32], next to *Signal* and *Timer*, is one of so-called event classes. All operations of a class can be obtained via *behavioralFeature* association (see **Figure 6.21**). Some of these operations may be constructors (*constructor* association) or destructors (*destructor* association). UML standard defines a constructor as "any operation having a single return result parameter of the type of the owning class [OMG 2004]." On the contrary, constructors in TAU are operations having the same names as classes to which they belong, and that do not have any explicit result parameters. A *Class* may have one default constructor accessible via *parameterlessConstructor* association.

[32] For the taxonomy of *Operation* metaclass refer to Appendix G.

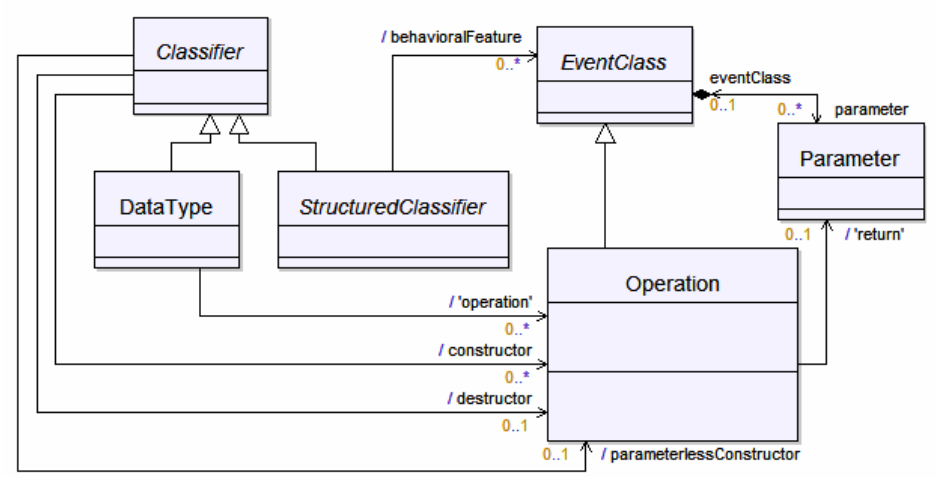

Figure 6.21. *Operation* metaclass in the role of an operation of a class

All parameters of an operation can be obtained via *parameter* association, and the return one – using *return* association. In operation bodies, they can be referenced/accessed via *definitions* indicated by instances of *Ident* metaclass.

6.9.2.1 Operation invocation

An invocation of an operation of a passive class that is not a constructor can be modelled with the use of *CallExpr* that has two important attributes – *called* pointing the called operation, and *to* optionally indicating the *Ident* of an attribute on which the operation is invoked (see **Figure 6.22**). As *CallExpr* inherits from *ActualArgumentContainer*, it enables invocation of operations with actual arguments accessible via *argument* association.

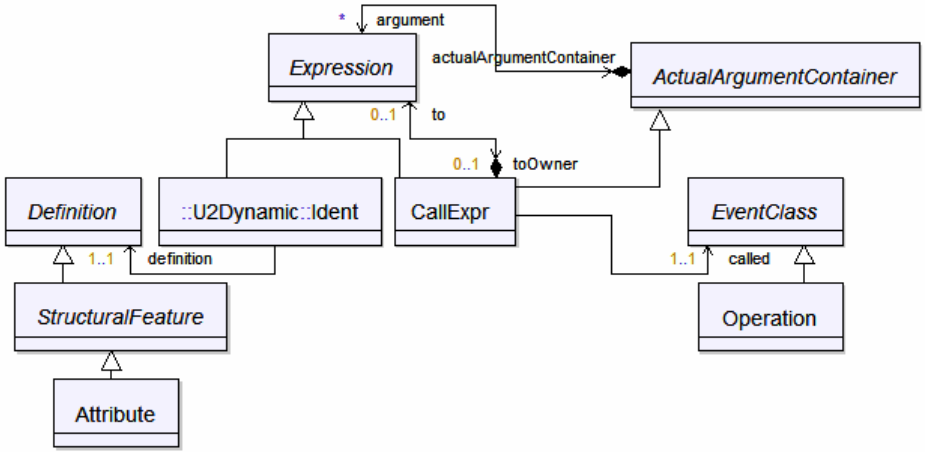

Figure 6.22. *CallExpr* – a fragment of TAU Object Model

A *CallExpr* can be contained directly in an *ExpressionAction* or indirectly in another expression or action in a *CompoundAction* representing either a body of an operation in a passive or an active class or actions on a triggered transition in a state machine of an active one. It is noteworthy that in the latter case the transition can connect two states in a state machine nested in another state.

6.10 Specification of TAU Executable UML Refactorings

As indicated by Staroń & Kuźniarz [2004], there are many different ways of defining UML model transformations, but in a practical approach, a specification of such a transformation consists of two parts – an informal one and a formal one. In the former, basic ideas behind a transformation are expressed, usually in the natural language, and the latter formalizes (usually in OCL) both conditions for the allowable usage of the transformation (preconditions) and its obligations (postconditions). The informal part makes the transformation more comprehendible, and the formal one facilitates its implementation. Moreover, as stated by Kazato *et al.* [2004], formalization of refactoring transformations is a necessary step on the way to their automation.

Each refactoring in this thesis is specified according to the following template:

1. **Informal specification**:
    a. **Name** — the name of a model refactoring
    b. **Origin** (optional) — the name of a code refactoring from which it is derived, and a reference to its description – only if it is derived from a code refactoring
    c. **Areas** — the refactoring area(s) of the transformation
    d. **Trigger-element** — the trigger-element of the transformation
    e. **Definitions** — a list of definitions used in the informal specification – each definition has a name and an explanation of its meaning; the symbol [T] after the name indicates that the definition is a trigger-element; in specifications, definitions are referred to via their names in curly brackets
    f. **Aim** — the goal of the transformation (one sentence)
    g. **Reasons** — probable reason(s) for performing the refactoring
    h. **Description** (optional) — a short explanation of the refactoring – only if its intent is not obvious
    i. **Bad smell** (optional) — a description of a bad smell tightly coupled with the refactoring, i.e. the one that can be usually removed with the use of the transformation
    j. **Preconditions** — a list of requirements that have to be fulfilled by a model in order to enable an execution of the transformation, as well as their explanation and justification
    k. **Postconditions** — a list of properties that have to be satisfied by the model in order to approve the refactoring – these conditions are based on the frame assumption, i.e. all modifications are limited to the ones mentioned in postconditions
    l. **Mechanics** — a mechanics of the transformation – identification of *basic operations* and/or other refactorings and the order in which they should be applied to cause fulfilment of the postconditions
    m. **Algorithm** — an algorithm of the transformation outlined in section 1.l, expressed with the use of basic operations and/or other refactorings

2. **Formal specification**:
    a. **Signature** — the transformation heading in the OCL-like format

b. **Bad smell** The bad smell from section 1.i
c. **Preconditions** the preconditions from section 1.j expressed in OCL
d. **Postconditions** the postconditions from section 1.k expressed in OCL

The format of the transformation signature (2.a) is following:

```
context ModelElement::RefactoringName(ArgumentsList),
```

where ModelElement is a metaclass of TAU Object Model being the type of the context element, RefactoringName is the unique name of a refactoring (written without spaces), and ArgumentsList is a list of any number of transformation formal parameters, which define existing model elements or some properties – usually names – of model elements that will be created and added to the model during the transformation. Each argument is specified in the following way:

```
ArgumentName : ArgumentType,
```

where ArgumentName is a symbolic name of the parameter that can be later used in pre- and postconditions, and ArgumentType is a TAU Object Model metaclass being the type of this parameter. Arguments are separated from each other with comas.

6.10.1 Basic Operations

As already observed by Kazato *et al.* [2004], each refactoring can be expresses as a sequence of *basic operations* that introduce (1) modifications that are essential for a given refactoring, and (2) changes that ensure that an output model will be consistent and correct.

In this thesis, two kinds of basic operations are distinguished, namely (1) *structural basic operations* that modify *structural model elements*, and (2) *behavioural basic operations* that modify *behavioural model elements*. If a basic operation seems to be both structural and behavioural, then is should be split into two parts – one structural and one behavioural.

Some basic operations have the same names and parameters as refactorings. To distinguish between them, the names of the latter ones are capitalized (e.g. renameClass and RenameClass). The main difference between basic operations and refactorings is that the latter ones are behaviour preserving. In practice, it means that e.g. renameAttribute (basic operation), as opposed to RenameAttribute (refactoring), can give an attribute defined in a class ClA a name that is already used by another one in ClA. If needed, a refactoring can be a part of the algorithm of another refactoring, what reflects the presence of "includes" relationship between these transformations.

In the frames of the work on an initial catalogue of TAU executable UML model refactorings (see Section 7), following (in alphabetical order) basic operations[33] have been identified:

```
context Class::renameClass(newName: String)
post: name = newName

context Operation::renameOperation(newName: String)
post: name = newName
```

[33] Their specifications are provided in OCL [Warmer & Kleppe 1999].

```
context Class::addGetter(a: Attribute)
post: behavioralFeature->exists(o: Operation | o.isGetter(a))
/* query isGetter is defined in Section 7.1.3 */

context CallExpr::replaceInvocation(accessor: Operation, o: Operation)
post: called = o and to.notEmpty() and to.oclIsTypeOf(CallExpr) and to.called =
accessor and to.to = to@pre)

context Operation::removeOperation()
post: not self@pre.namespace.behavioralFeature->exists(o: Operation | o =
self@pre)

context Attribute::renameAttribute(newName: String)
post: name = newName

context Attribute::moveAttribute(from: Class, to: Class)
post: not from.attribute->exists(a: Attribute | a = self) and to.attribute-
>exists(a: Attribute | a = self)

context Attribute::removeAttribute()
post: not self@pre.namespace.attribute->exists(o: Attribute | o = self@pre)

context Class::addAttribute(a: Attribute)
post: attribute->exists(attr: Attribute | attr.name = a.name and attr.type =
a.type and attr.visibility = a.visibility)

context Operation::changeVisibility(vk: VisibilityKind)
post: visibility = vk

context Operation::appendParameter(n: String, t: Type)
post: parameter->exists(p: Parameter | p.name = n and p.type = t)

context CallExpr::addDefaultValueForParameter(n: String)
post: let pos = called.parameter->iterate(p: Parameter; position: Integer = 0 |
if p.name<>n then position+1 else position endif) in
let arg = argument->at(pos) in
arg.oclIsTypeOf(Ident) and arg.definition.oclIsTypeOf(Literal) and
arg.definition.name = 'NULL'

context Parameter::removeParameter()
post: not self@pre.namespace.parameter->exists(p: Parameter | p = self@pre)
```

7 INITIAL CATALOGUE OF TAU EXECUTABLE UML MODEL REFACTORINGS

The catalogue contains twelve specifications of exemplary TAU executable UML model refactorings triggered on the following elements from ESPC and ESAC areas:

- Class
- Class::Attribute
- Class::Operation
- Class::Operation::Parameter

The majority of presented transformations are derived from Fowler's catalogue [Fowler *et al.* 1999]. All refactorings are specified in accordance with the template introduced in Section 6.10. They are assumed to work properly only on a model that can be compiled without any errors and warnings to the Model Verifier.

By an *inheritance hierarchy* of class ClA understood are all – both direct as well as indirect – subclasses and superclasses of ClA. A term *sibling* should be understood in the following way:

- A *sibling operation* of operation opA – an operation having the same name and the same non-return parameters as opA;
- A *sibling attribute* of attribute atA – an attribute having the same name and the same type as atA.

Formal specifications of transformations are written in OCL [Warmer & Kleppe 1999]. Queries that are used in more than one specification are listed (in alphabetical order) and defined in Section 7.5.

In the opinion of the authors, the catalogue will be useful for both model designers and maintainers as well as for vendors of UML CASE tools.

7.1 Trigger-element – Class

This section contains specifications of following refactorings:

1. Rename Class (Passive)
2. Rename Class (Active)
3. Remove Middle Man

7.1.1 Rename Class (Passive)

Informal specification

Areas: ESPC
Trigger-element: PassiveClass

Definitions:

1. class [T] – a class that is to be renamed
2. newName – a new name of {class}
3. constructor – a constructor of {class}
4. destructor – a destructor of {class}

Aim: Change the name of {class} to {newName}.

Reasons: The current name of {class} does not reflect its purpose.

Preconditions:

1. {class} is passive.
2. {newName} adheres to TAU naming rules.
3. There is no classifier with the name {newName} in the namespace in which {class} is defined.

The precondition results from the UML constraint stating that all the members of a namespace are distinguishable within it. The default rule is that two elements are distinguishable if they have unrelated types, or related types but different names [OMG 2004]. Both the constraint and the rule are valid in TAU[34].

4. In {class} there is no operation with the name {newName} and the same parameters as any {constructor}.

The precondition results from the same constraint and the same rule as the first one. In this case, the transformation has to assure that new signatures[35] of any {constructors} named after {class} will not conflict with signatures of operations that are already defined in it.

5. There is no class with the name {newName} in any class defined in the same namespace as {class}.

This precondition is necessary due to the TAU binding mechanism that may cause a situation like the one shown in **Figure 7.1**, where after renaming *Class1* to *Class3*, *Class1* in create expression in *Class2.op1()* is rebound to *Class3* nested in *Class2*.

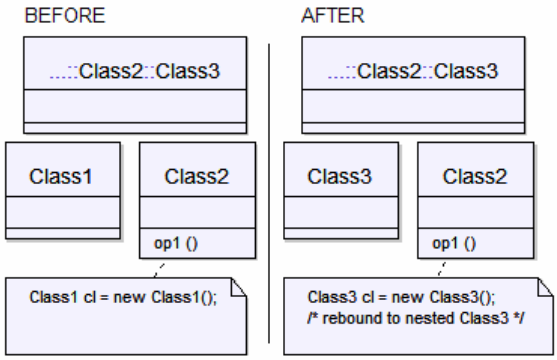

Figure 7.1. Rename of a class and binding mechanism

Postconditions:

1. The name of {class} is {newName}.
2. All references to {class} are via {newName}.
3. Names of all {constructors} are {newName}.
4. All invocations of all {constructors} are via {newName}.
5. Names of all {destructors} are {~newName}.
6. All invocations of all {destructors} are via {~newName}.

[34] Additionally, in TAU each model element is distinguishable by a Globally Unique Identifier (GUID) that remains unchanged for its entire lifetime.
[35] Operations are distinguished by their signatures, i.e. their names and non-return parameters.

Mechanics

The first step in the mechanics of Rename Class is a single invocation of *renameClass* – this causes fulfilment of the first two preconditions. Additionally, this operation causes automatic adjustment of invocations of all {constructors} (the fourth precondition). However, this modification is not accompanied by rename of definitions of {constructors}, what is highly undesired. It is difficult to explain this phenomenon by investigating TAU Object Model, because it turns out that a create expression references a constructor, and not a class, which instance is intended to be created. Nevertheless, it can lead to a situation like the one depicted in **Figure 7.2**, where all invocations of an overloaded default constructor *ClassA()*, that e.g. initializes an attribute *i*, are replaced by invocations of an implicit one (e.g. in the state machine of *ClassB*). This results in the change in behaviour that is not detected by the component responsible for model checking - the attribute *i* is not initialized during creation of an instance of *ClassC*.

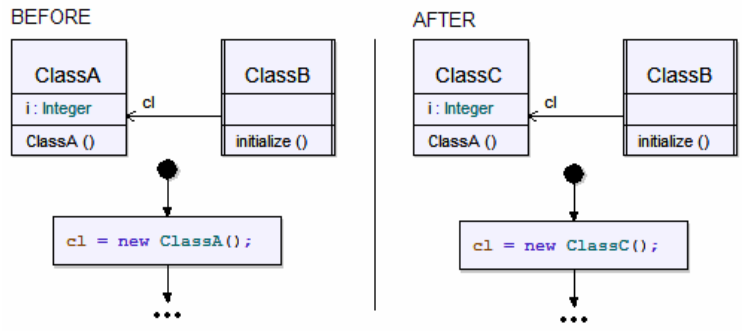

Figure 7.2. Rename of a class not accompanied by rename of constructors

Therefore, the second step of the transformation is to invoke *renameOperation* for all {constructors}. Moreover, since in the effect of *renameClass* names of {destructors} are updated neither in their definitions nor in their explicit invocations, *renameOperation* should be invoked also for all {destructors}.

Algorithm:

1. *{class}.renameClass({newName})*
2. for each {constructor} – *{constructor}.renameOperation({newName})*
3. for each {destructor} – *{destructor}.renameOperation(~{newName})*

Formal specification

```
context Class::RenameClass(newName: String)

pre : not isActive --1
and newName.isValidName() --2
and not namespace.ownedMember->exists(c: Classifier | c.name = newName) --3
and not behavioralFeature->exists(o: Operation | o.name = newName and
o.hasTheSameParameters(constructor)) --4
and not namespace.ownedMember->exists(c: Class | c.ownedMember->exists(cl:
Class | cl.name = newName)) --5

post: name = newName --1&2
and behavioralFeature->forAll(o: Operation | constructor@pre->includes(o)
implies o.name = newName) --3&4
and behavioralFeature->forAll(o: Operation | destructor@pre->includes(o)
implies o.name = '~'+newName) --5&6
```

7.1.2 Rename Class (Active)

Informal specification

Areas: ESAC
Trigger-element: ActiveClass
Definitions – unchanged
Aim – unchanged
Reasons – unchanged

Preconditions:

1. {class} is active.
2. Unchanged
3. Unchanged
4. There is no event class with the name {newName} and the same parameters as any constructor named after {class} in the namespace determined by {class}.
5. Unchanged

Postconditions:

1. Unchanged
2. Unchanged
3. Names of all constructors, besides a state machine named *initialize* (if such exists), of {class} are {newName}.
4. Unchanged
5. Invalid
6. Invalid

Mechanics

Renamed are all constructors of {class} except for a state machine named *initialize* (if such exists). As active classes do not support destructors, they are not renamed by the refactoring.

Algorithm:

1. Unchanged
2. for each constructor (besides *initialize* state machine) of {class} – *{constructor}.renameOperation({newName})*
3. Invalid

Formal specification

```
context Class::RenameClass(newName: String)

pre : isActive --1
and (…) --2&3 unchanged
and not behavioralFeature->exists(ec: EventClass | ec.name = newName and
ec.hasTheSameParameters(constructor)) --4
and (…)--5 unchanged

post: (…) --1&2 unchanged
and behavioralFeature->forAll(o: Operation | (o.oclIsTypeOf(StateMachine) and
o@pre.name <> 'initialize') and constructor@pre->includes(o) implies o.name =
newName) --3&4; 5&6 invalid
```

7.1.3 Remove Middle Man

Informal specification

Origin: Remove Middle Man [Fowler *et al.* 1999]
Areas: ESPC/ESAC
Trigger-element: PassiveClass/ActiveClass

Definitions:

1. middleMan [T] – a class being a "middle man"
2. delegate – a class being a "delegate"
3. delegateAttribute – an attribute of {middleMan} typed by {delegate}
4. delegatingOperation – an operation in {middleMan} which only invokes {delegatedOperation} via {delegateAttribute}
5. delegatedOperation – an operation in {delegate} which is invoked by {delegatingOperation}
6. getter – accessor for {delegateAttribute}

Aim: Stop {middleMan} from being a mediator for {delegate}

Reasons: {middleMan} is doing too much simple delegation

Description

As stated by Fowler, "one of the keys, if not *the* key, to objects is encapsulation [Fowler *et al.* 1999]." In general, the less each class in a model needs to know about other classes, the less possible is that a change in one place causes the necessity to adjust other parts of the model, what makes the model maintenance easier and cheaper. For instance, let us consider a situation in which a client class invokes an operation defined on one of the attributes of a server class. As the client has to know about the delegate class, each change of the delegate may propagate to the client. This redundant dependency can be removed by placing a simple delegating operation on the server, which hides the delegate. Now, changes become limited to the server and do not propagate to the client. However, as observed by Fowler [ibid.], the price for this encapsulation is that every time the client wants to use a new feature of the delegate, another delegating method has to be added to the server what can become awkward. Is such a situation, it may be convenient to apply on the server Remove Middle Man refactoring.

A part of an exemplary model illustrating Remove Middle Man is shown in **Figure 7.3**. Before the transformation, *MiddleMan* gives *Client* the use of two simple delegating operations – *delegatingOp1()* and *delegatingOp2()* – that do nothing else besides invoking appropriate operations on *Delegate* – *delegatedOp1()* and *delegatedOp2()* respectively. The refactoring removes both delegating operations, introduces *getDelegate()* accessor for *delegate* attribute in *MiddleMan*, and replaces all invocations of deleted delegating operations by calls of corresponding delegated ones via the accessor. The main benefits of the application of the transformation are (1) reduced number of operations in *MiddleMan*, and (2) improved communication.

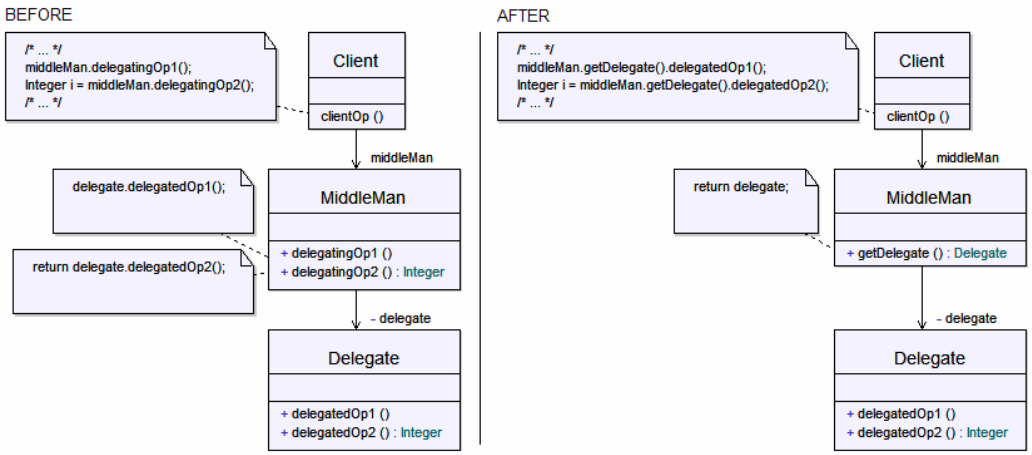

Figure 7.3. Remove Middle Man – an example

This version of Remove Middle Man works for a pair of classes *A* and *B*, where *A* is a middle man of a delegate *B*. However, even after performing the refactoring for a middle man *A* and its delegate *B*, *A* can still be a middle man for other delegates. If needed, the transformation can be repeated for any pair consisting of *A* and a delegate class until *A* stops being a middle man at all.

Bad smell

A bad smell directly associated with the refactoring is called *Middle Man*. Wake [2003] defines its symptom in the following way: "most methods of a class call the same or a similar method on another object." Usually, it may occur in a model from applying Hide Delegate to address another bad smell called *Message Chains* that trades off against *Middle Man*. One should be careful in removing middle men, because they may be intentionally created by some design patterns (e.g., Proxy or Decorator). Moreover, as middle men provide a sort of façade, removing them can expose clients to more information than they should know [ibid.].

In the context of this bad smell, a middle man for a delegate can be defined as a class that has at least two simple delegating operations for any delegated operations of this delegate. A simple delegating operation is characterised by its body that has only one statement in the form of either

delegate.delegatedOp() or return delegate.delegatedOp(),

where *delegate* is an attribute typed by potential delegate class, and *delegatedOp()* is one of its operations. Obviously, in the latter case, the return parameter of a delegating operation has the same type as *delegatedOp()*. The repository view [Bock 2003] of the operation body of the delegating operation *delegatingOp1()* owned by the class *MiddleMan*, which is introduced in the example illustrated in **Figure 7.3**, is shown in **Figure 7.4**.

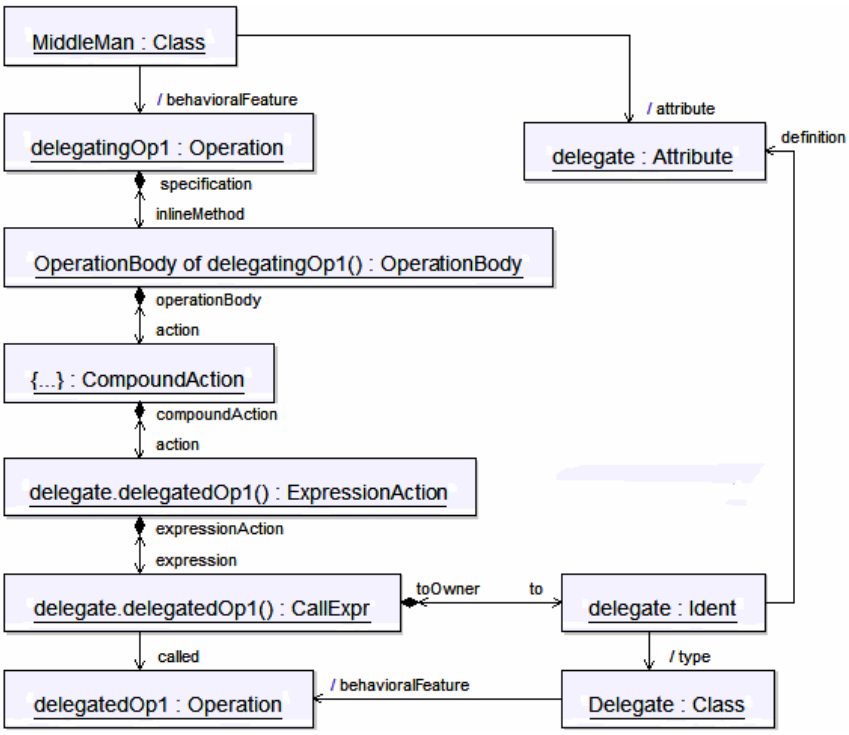

Figure 7.4. Repository view of the body of a simple delegating operation

The body consists of one *CompoundAction* that in turn contains an *ExpressionAction* having a *CallExpression*, which owns an *Ident*, typed by *Delegate* class, pointing the *delegate* attribute of *MiddleMan* class. Additionally, the *CallExpression* indicates invoked *delegatedOp1()* that belongs to *Delegate*. The repository view of the value-returning version of a delegating operation differs from the non-returning one in the following way – the *ExpressionAction* is replaced by a *ReturnAction*, and the *CallExpresion* has an additional attribute *type* indicating the type of the return parameter (see **Figure 7.5**).

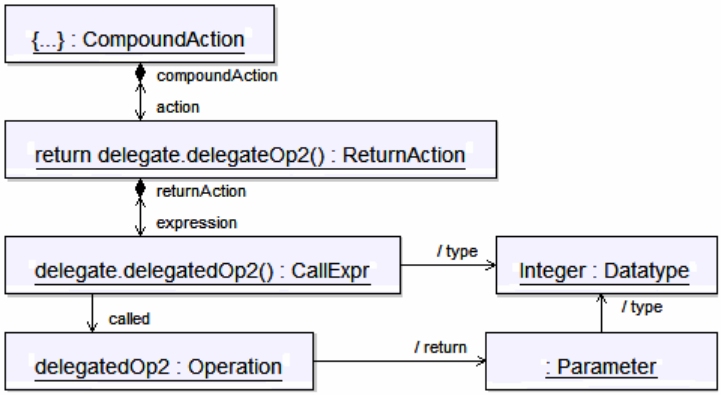

Figure 7.5. Differences between a value-returning and non-returning operation

An extracted fragment of TAU Object Model that corresponds to the examples of the bodies of delegating operations is presented in **Figure 7.6**.

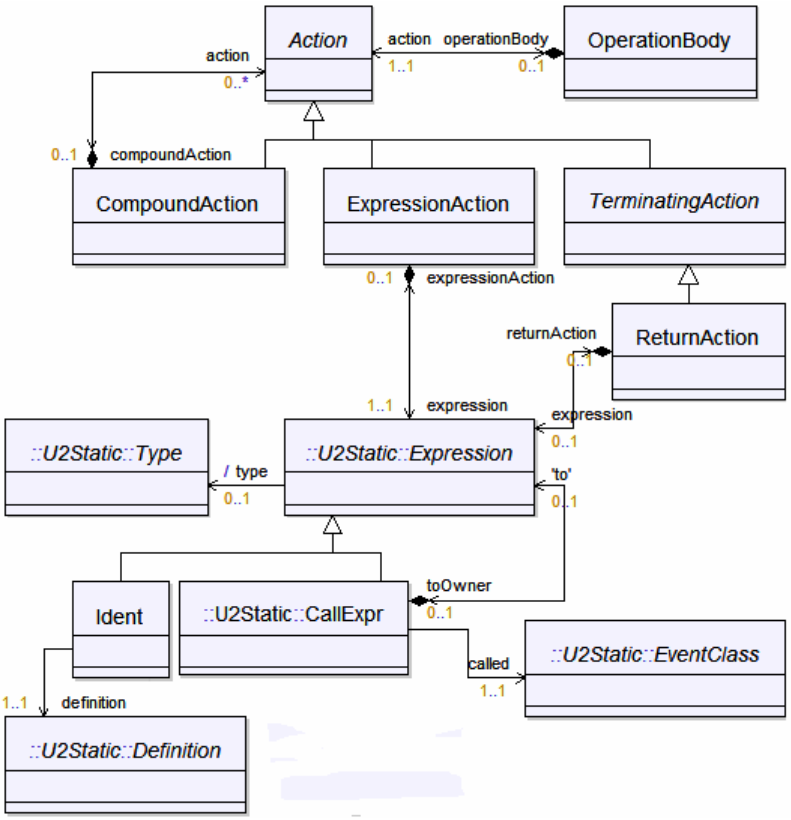

Figure 7.6. Fragment of TAU Object Model – operation body of a delegating operation

To be able to automatically detect all potential middle men in a model, one has to define and implement a query checking whether a given class has at least two simple delegating operations for a particular attribute. Next, this query can be run for each pair consisting of a class (possible middle man) and its attribute typed by another class (possible delegate).

A simple delegating operation for an attribute *A* fulfils following conditions:

1. The non-value returning version:
    a. It has a body with a *CompoundAction*;
    b. The *CompoundAction* owned by the body contains one *ExpressionAction*;
    c. The *ExpressionAction* owns a *CallExpr*;
    d. The *to* attribute of the *CallExpr* points an *Ident*;
    e. The *definition* attribute of the *Ident* points *A*;
    f. The *called* attribute of the *CallExpr* indicates an *Operation* that is a behavioural feature of a *Class* typed by the *Ident*.
2. The value returning version:
    a. Unchanged;
    b. The *CompoundAction* owned by the body contains one *ReturnAction*;
    c. The *ReturnAction* owns a *CallExpr*;
    d. Unchanged;
    e. Unchanged;
    f. Unchanged;
    g. The *type* attribute of *CallExpr* points the same *Type* as the *type* attribute of the return parameter of the *Operation*.

The first version of the operation could have an additional condition stating that it has no parameters, and the second one – that it has one return parameter of the same type as the *Operation* (i.e. the delegated operation). However, it is not necessary, because the latter conditions enforce their fulfilment.

Preconditions:

1. {delegateAttribute} is an attribute of {middleMan}.
2. {delegateAttribute} is typed by a class.
3. {middleMan} has at least one {delegatingOperation} for {delegateAttribute}

Postconditions:

1. There is {getter} in {middleMan}.
2. Each invocation of each {delegatingOperation} is replaced by invocation of corresponding {delegatedOperation} via {getter}.
3. There is no {delegatingOperation} in {middleMan}.

Mechanics

The first step in the mechanics of Remove Middle Man is creation of an accessor for {delegateAttribute} – of course, only if it does not yet exist. Next, each invocation of each {delegatingOperation} is replaced by invocation of a corresponding {delegatedOperation} via just created {getter} (see **Figure 7.7**). Finally, each {delegatingOperation} should be removed from {middleMan}.

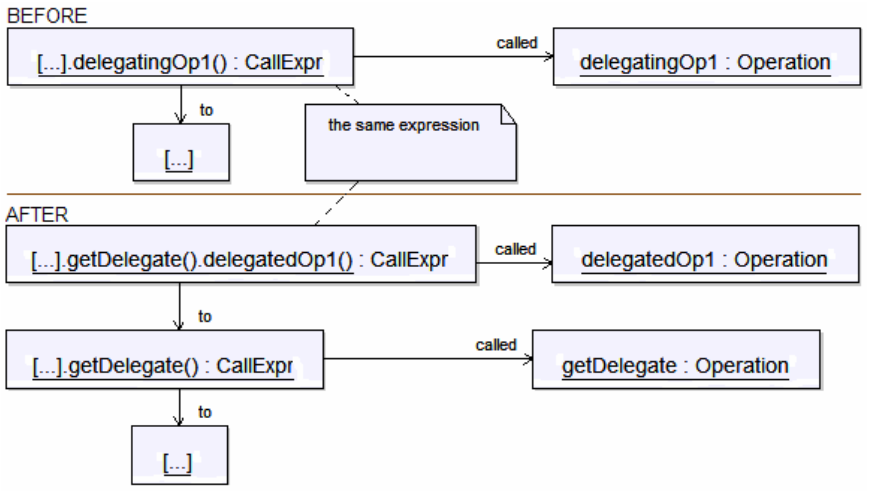

Figure 7.7. Replacement of invocations of delegating operations

Algorithm:

1. *{getter} = {middleMan}.addGetter({delegateAttribute})*
2. for each invocation of each {delegatingOperation} – *{invocationOfDelegatingOperation}.replaceInvocation({getter},{delegatedOperation})*
3. for each {delegatingOperation} – *{delegatingOperation}.removeOperation()*

Formal specification

```
context Class::RemoveMiddleMan(delegate: Attribute)

smell: isMiddleMan(delegate)
```

```
context Class::isMiddleMan(A: Attribute): Boolean
body: self.simpleDelegatingOperations(A)->size() >= 2

context Class::simpleDelegatingOperations(A: Attribute): Set(Operation)
body: behavioralFeature->select(o: Operation |
o.isSimpleDelegatingOperation(A))

context Operation::isSimpleDelegatingOperation(A: Attribute): Boolean
body: isNonValueReturningSDO(A) or isValueReturningSDO(A)

context Operation::isNonValueReturningSDO(A: Attribute): Boolean
body:
let ac = inlineMethod.action.action in
let ex = ac.expression in
inlineMethod->notEmpty() and inlineMethod.action->notEmpty() --a
and (ac->size() = 1 and ac->first().oclIsTypeOf(ExpressionAction)) --b
and ex.oclIsTypeOf(CallExpr) --c
and (ex.to->notEmpty() and ex.to.oclIsTypeOf(Ident)) --d
and ex.to.definition = A --e
and ex.called.oclIsTypeOf(Operation) --f
and ex.to.type.behavioralFeature->includes(ex.called) --g

context Operation::isValueReturningSDO(A: Attribute): Boolean
body:
let ac = inlineMethod.action.action in
let ex = ac.expression in
(…) --a: as in isNonValueReturningSDO
and (ac->size() = 1 and ac->first().oclIsTypeOf(ReturnAction)) --b
and (…) --from c to g: as in isNonValueReturningSDO
and ex.type->first() = ex.called.return.type.first() --h


pre : attribute->includes(delegate) --1
and delegate.type.oclIsTypeOf(Class) --2
and self.simpleDelegatingOperations(delegate)->size() >= 1 --3

post: behavioralFeature->exists(o: Operation | o.isGetter(delegate)) --1
and session.getAllCallExpr()->forAll(e: CallExpr |
e.called@pre.isSimpleDelegatingOperation(delegate) implies e.called =
e.called@pre.inlineMethod.action.action.expression.called and let e2 =
e.called.to in (e2->notEmpty() and e2.oclIsTypeOf(CallExpr) and
e2.called.isGetter(delegate) and e2.to = e.to@pre)) --2
and not behaviouralFeature->exists(o: Operation |
o.isSimpleDelegatingOperation(A)) --3

context Operation::isGetter(A: Attribute): Boolean
body:
let ac = inlineMethod.action.action in
let ex = ac.expression in
visibility = #VkPublic and name = 'get'.concat(A.name)
and parameter->size() = 1 and (return->notEmpty() and return.type = A.type)
and inlineMethod->notEmpty()
and (ac->size() = 1 and ac->first().oclIsTypeOf(ReturnAction))
and (ex->first().oclIsTypeOf(Ident) and ex.definition = A)
```

7.2 Trigger-element – Attribute of Class

This section contains specifications of following refactorings:

1. Rename Attribute (Passive)
2. Rename Attribute (Active)
3. Pull Up Attribute (Passive)
4. Push Down Attribute (Active)

7.2.1 Rename Attribute (Passive)

Informal specification

Areas: ESPC

Trigger-element: PassiveClass::Attribute

Definitions:

1. attribute [T] – an attribute that is to be renamed
2. class – a class in which {attribute} is defined
3. newName – a new name of {attribute}

Aim: Change the name of {attribute} defined in {class} to {newName}.

Reasons: The current name of {attribute} does not reflect its purpose.

Preconditions:

1. {class} is passive.
2. {newName} adheres to TAU naming rules.
3. There is no attribute with the name {newName} in the inheritance hierarchy of {class}.

Rename Attribute changes names of all attributes having the same name as {attribute} in the whole inheritance hierarchy of {class}. This precondition assures that there will be no name conflicts between renamed attributes and already existing ones.

4. There is no operation with a parameter or a local variable with the name {newName} in the inheritance hierarchy of {class}.

Operations can read and write attributes of classes to which they belong. Therefore, after renaming an attribute to a name of a parameter or a local variable of an operation, each reference to the attribute in the body of the operation is immediately rebound to the parameter/variable. The Model Checker detects this situation only if types of the attribute and the parameter/variable are incompatible.

Postconditions:

1. The name of {attribute} is {newName}.
2. All references to {attribute} are via {newName}.
3. Names of all attributes with the same name as {attribute} defined in the inheritance hierarchy of {class} are {newName}.
4. All references to all attributes with the same name as {attribute} defined in the inheritance hierarchy of {class} are via {newName}.

Mechanics

The first step in the mechanics of Rename Attribute is a single invocation of *renameAttribute* – this causes fulfilment of the first two preconditions. If {class} has neither super- nor subclasses with sibling attributes of {attribute}, then the refactoring finishes. Otherwise, it is necessary to rename also these sibling attributes, what satisfies the latter two preconditions. This transformation step bases on the assumption that attributes in an inheritance hierarchy having the same names are semantically related, and thus they should evolve together. Moreover, omission of this step could lead to a situation like the one shown in **Figure 7.8**[36], where after renaming *a1* to *a2* in *ClassB*, reference to *a1* in *ClassC* causes access of *a1* from *ClassA*, instead of *a1* from *ClassB*, as it was before refactoring[37].

[36] It is noteworthy that the binding mechanism of TAU prevents from this threat.
[37] Assuming that *a1* in *ClassB* does not have private visibility.

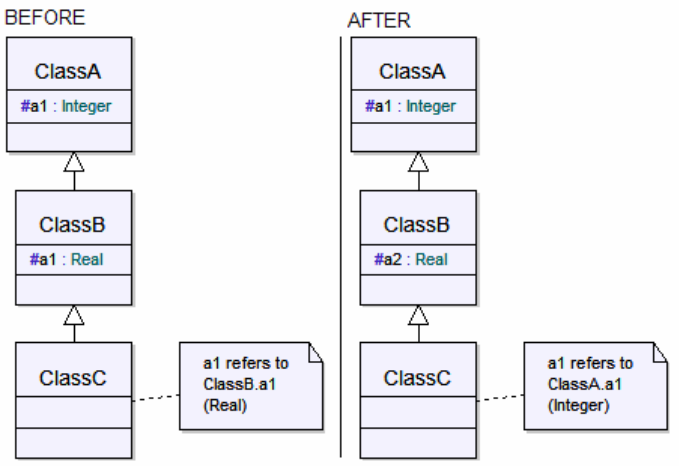

Figure 7.8. Rename of an attribute not accompanied by rename of its siblings

Algorithm:

1. *{attribute}.renameAttribute({newName})*
2. for each sibling attribute of {attribute} – *{sibling_attribute}.renameAttribute({newName})*

Formal specification

```
context Attribute::RenameAttribute(newName: String)

pre : not isActive --1
and newName.isValidName() --2
and let allClasses = namespace.getAllClassesFromInheritanceHierarchy() in
not allClasses.attribute->exists(a: Attribute | a.name = newName) --3
and not allClasses.behavioralFeature->exists(o: Operation |
o.hasParameterNamed(newName) or o.hasLocalVariableNamed(newName)) --4

post: name = newName --1&2
and namespace.getAllClassesFromInheritanceHierarchy().attribute->forAll(a:
Attribute | a.name@pre = self.name@pre implies a.name = newName) --3&4

context Operation::hasParameterNamed(n: String): Boolean
body: parameter->exists(p: Parameter | p.name = n)
```

7.2.2 Rename Attribute (Active)

Informal specification

Areas: ESAC
Trigger-element: ActiveClass::Attribute
Definitions – unchanged
Aim – unchanged
Reasons – unchanged

Preconditions:

1. {class} is active.
2. Unchanged
3. Unchanged
4. Unchanged
5. {attribute} is not declared in any interface realized by any class in the inheritance hierarchy of {class}.

The precondition prevents from renaming attributes that implement the corresponding ones declared in realized interfaces. It reduces the scope of change of the refactoring. Otherwise, it would be also necessary to rename the attributes in these interfaces, what would cause the need for renaming attributes in all inheritance hierarchies of classes that realize these interfaces. An example of such a chain of changes is shown in **Figure 7.9**, where renaming *a1* to *a3* in *Class1* enforces renaming *a1* in *Class2*, *Interface1*, *Class3*, and *Class4*. In the opinion of the authors of the catalogue, this is another refactoring (called e.g. Rename Interface Attribute) that is triggered on an attribute contained in an interface, and not in a class.

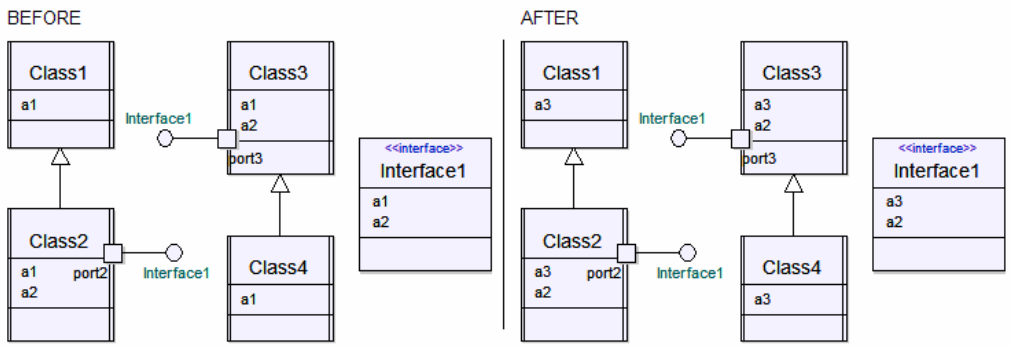

Figure 7.9. Rename Attribute - change propagation through an interface

Postconditions – unchanged
Mechanics – unchanged
Algorithm – unchanged

Formal specification

```
context Attribute::RenameAttribute(newName: String)

pre : isActive --1
and (…) --2&3&4 unchanged
and not
namespace.getAllClassesFromInheritanceHierarchy().port.realized.attribute-
>exists(a: Attribute | a.name = newName) --5

post: (…) --1&2&3&4 unchanged
```

7.2.3 Pull Up Attribute (Passive)

Informal specification

Origin: Pull Up Field [Fowler *et al.* 1999]
Areas: ESPC
Trigger-element: PassiveClass::Attribute

Definitions:

1. attribute [T] – an attribute that is to be pulled up
2. class – a class in which {attribute} is defined
3. superclass – a superclass of {class}
4. subclasses – all direct subclasses of {superclass} except for {class}

Aim: Move {attribute} and its siblings from all {subclasses} to {superclass}

Reasons: {attribute} has siblings in all {subclasses}

Preconditions:

1. {class} is passive.
2. {class} has a superclass.
3. In all {subclasses} defined is a sibling of {attribute}.
4. There is no attribute with the name of {attribute} in {superclass}.

Postconditions:

1. {attribute} is defined in {superclass}.
2. Siblings of {attribute} are not defined in {subclasses}.

Mechanics

First, {attribute} is moved from {class} to {superclass}. Next, all siblings of {attribute} are removed from {subclasses}.

Algorithm:

1. *{attribute}.moveAttribute({class},{superclass})*
2. for each sibling of {attribute} – *{sibling_attribute}.removeAttribute()*

Formal specification

```
context Attribute::PullUpAttribute()

pre : not namespace.isActive --1
and namespace.supertype->notEmpty() --2
and namespace.subtypes->forAll(c: Class | c.attribute->exists(a: Attribute |
a.name = self.name and a.type = self.type)) --3
and not namespace.supertype.attribute->exists(a: Attribute | a.name =
self.name) --4

post: namespace=namespace@pre.supertype --1
and namespace.subclasses->forAll(c: Class | not c.exists(a: Attribute | a.name
= self.name and a.type = self.type)) --2
```

7.2.4 Push Down Attribute (Passive)

Informal specification

Origin: Push Down Field [Fowler *et al.* 1999]
Areas: ESPC
Trigger-element: PassiveClass::Attribute

Definitions:

1. attribute [T] – an attribute that is to be pushed down
2. class – a class in which {attribute} is defined
3. subclasses – all direct subclasses of {class}

Aim: Move {attribute} to only these {subclasses} that use it

Reasons: {attribute} is used only by some {subclasses}

Preconditions:

1. {class} is passive.
2. {attribute} is neither read nor written in/through {class}.
3. No {subclass} contains an attribute with the same name as {attribute}.

Postconditions:

1. {attribute} is not defined in {class}.
2. Siblings of {attribute} are defined in {subclasses} that use it.

Mechanics

First, a copy of {attribute} is added to each {subclass} that needs it. Next, {attribute} is removed from {class}.

Algorithm:

1. for each {subclass} that needs {attribute} –
   {subclass}.addAttribute({attribute})
2. *{attribute}.removeAttribute()*

Formal specification

```
context Attribute::PushDownAttribute()

pre : not isActive --1
and session.getAllFieldExpr()->forAll(fe: FieldExpr | fe.field = self implies
fe.expression.type <> self.namespace) and namespace.behavioralFeature-
>forAll(o: Operation | not o.inlineMethod.getAllIdent()->exists(i: Ident |
i.definition = self)) --2
and namespace.namespace.allSubclasses->forAll(c: Class | not c.attribute-
>exists(a: Attribute | a.name = self.name)) --3

post: not namespace@pre.attribute->exists(a: Attribute | a = self@pre) --1
and namespace@pre.subclasses->select(c: Class |
self.session@pre.getAllFieldExpr()->exists(fe: FieldExpr | fe.field = self and
fe.expression.definition.type = c) or c.behavioralFeature->exists(o: Operation
| o.inlineMethod.getAllIdent()->exists(i: Ident | i.definition = self)))-
>forAll(c: Class | c.attribute->exists(a: Attribute | a.name = self.name and
a.type = self.type)) --2

context Session::getAllFieldExpr(): Set(FieldExpr)
/* the query returns a set containing all instances of FieldExpr in the model
*/
```

7.3 Trigger-element – Operation of Class

This section contains specifications of following refactorings:

1. Rename Operation (Passive)
2. Rename Operation (Active)
3. Hide Operation (Passive)
4. Add Parameter to Operation (Passive)

7.3.1 Rename Operation (Passive)

Informal specification

Origin: Rename Method [Fowler *et al.* 1999]
Areas: ESPC
Trigger-element: PassiveClass::Operation

Definitions:

1. operation [T] – an operation that is to be renamed
2. class – a class in which {operation} is defined
3. newName – a new name of {operation}

Aim: Change the name of {operation} defined in {class} to {newName}.

Reasons: The current name of {operation} does not reflect its purpose.

Preconditions:

1. {class} is passive.
2. {newName} adheres to TAU naming rules.
3. There is no operation with the name {newName} and the same non-return parameters as {operation} in the inheritance hierarchy of {class}.

Rename Operation changes names of all operations having the same names and the same parameters as {operation} in the whole inheritance hierarchy of {class}. This precondition assures that there will be no conflicts between signatures of renamed operations and other ones.

4. {operation} is neither a constructor nor a destructor.

Names of both constructors and destructors are strictly determined by names of classes in which they are defined. Therefore, they cannot be renamed separately from their containers.

Postconditions:

1. The name of {operation} is {newName}.
2. All references to {operation} are via {newName}.
3. Names of all operations with the same signature as {operation} defined in the inheritance hierarchy of {class} are {newName}.
4. All references to all operations with the same signature as {operation} defined in the inheritance hierarchy of {class} are via {newName}.

Mechanics

The first step in the mechanics of Rename Operation is a single invocation of *renameOperation* – this causes fulfilment of the first two preconditions. If {class} has neither super- nor subclasses with sibling operations of {operation}, then the refactoring finishes. Otherwise, it is necessary to rename also these sibling operations, what satisfies the latter two preconditions. This transformation step bases on the assumption that operations in an inheritance hierarchy that have the same signatures are semantically related, and thus they should evolve together. Moreover, omission of this step could lead to a situation like the one shown in **Figure 7.10**[38], where after renaming *op1()* to *op2()* in *ClassB*, invocation of *op1()* in *ClassC* causes call of *op1()* from *ClassA*, instead of *op1()* from *ClassB*, as it was before refactoring[39].

[38] It is noteworthy that the binding mechanism of TAU prevents from this threat.
[39] Assuming that *op1()* in *ClassB* does not have private visibility.

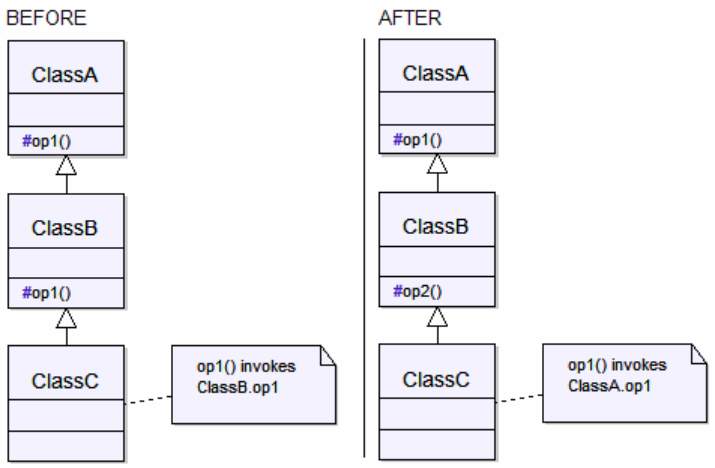

Figure 7.10. Rename of an operation not accompanied by rename of its siblings

Algorithm:

1. *{operation}.renameOperation({newName})*
2. for each sibling operation of {operation} – *{sibling_operation}.renameOperation({newName})*

Formal specification

```
context Operation::RenameOperation(newName: String)

pre : not isActive --1
and newName.isValidName() --2
and not namespace.getAllClassesFromInheritanceHierarchy().behavioralFeature-
>exists(o: Operation | o.name = newName and o.hasTheSameParameters(self)) --3
and operationKind <> #OkConstructor and operationKind <> #OkDestructor --4

post: name = newName --1&2
and namespace.getAllClassesFromInheritanceHierarchy().behavioralFeature-
>forAll(o: Operation | o.name@pre = self.name@pre and
o.hasTheSameParameters(self) implies o.name = newName) --3&4
```

7.3.2 Rename Operation (Active)

Informal specification

Areas: ESAC
Trigger-element: ActiveClass::Operation
Definitions – unchanged
Aim – unchanged
Reasons – unchanged

Preconditions:

1. {class} is active.
2. {newName} adheres to TAU naming rules.
3. There is no event class with the name {newName} and the same non-return parameters as {operation} in the inheritance hierarchy of {class}.
4. {operation} is not a constructor.
5. {operation} is not declared in any interface realized by any class in the inheritance hierarchy of {class}.

The precondition prevents from renaming operations that implement the corresponding ones declared in realized interfaces. It reduces the scope of change of

the refactoring. Otherwise, it would be also necessary to rename the operations in these interfaces, what would cause the need for renaming operations in all inheritance hierarchies of classes that realize these interfaces. An example of such a chain of changes is shown in **Figure 7.11**, where renaming *op1()* to *op3()* in *Class1* enforces renaming *op1()* in *Class2*, *Interface1*, *Class3*, and *Class4*. In the opinion of the authors of the catalogue, this is another refactoring (called e.g. Rename Interface Operation) that is triggered on an operation contained in an interface, and not in a class.

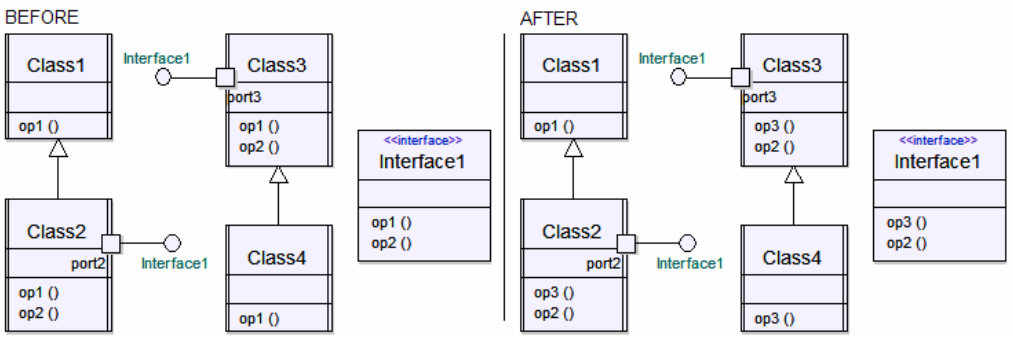

Figure 7.11. Rename Operation – Change propagation through an interface

Postconditions – unchanged
Mechanics – unchanged
Algorithm – unchanged

Formal specification

```
context Operation::RenameOperation(newName: String)

pre : isActive --1
and (…) ---2 unchanged
and not namespace.getAllClassesFromInheritanceHierarchy().behavioralFeature-
>exists(ec: EventClass | ec.name = newName and ec.hasTheSameParameters(self)) -
-3
and operationKind <> #OkConstructor --4
and not
namespace.getAllClassesFromInheritanceHierarchy().port.realized.behavioralFeatu
re->exists(o: Operation | o.name = newName and o.hasTheSameParameters(self)) --
5

post: (…) --1&2&3&4 unchanged
```

7.3.3 Hide Operation (Passive)

Informal specification

Origin: Hide Method [Fowler *et al.* 1999]
Areas: ESPC
Trigger-element: PassiveClass::Operation

Definitions:
1. operation [T] – an operation that is to be hidden
2. class – a class in which {operation} is defined
3. subclasses – all subclasses of {class}

Aim: Hide {operation} defined in {class} from all classes except for {subclasses}

Reasons: Only {class} and {subclasses} should be allowed to invoke {operation}

Description

The refactoring hides not only {operation} but also all its siblings in {subclasses}.

Preconditions:

1. {class} is passive.
2. {operation} is invoked only by {class} or {subclasses}.
3. Siblings of {operation} in {subclasses} are invoked only by their owners or owners' subclasses.

Postconditions:

1. If {class} has no subclasses or visibility of {operation} was *private*, then its visibility is *private*, else it is *protected*.
2. *Private* siblings of {operation} in {subclasses} remain *private* – the rest of them have *protected* visibility.

Mechanics

As the realization of the transformation relies on changing visibility of operations, it uses only one basic operation – *changeVisibility* – applied to {operation} and all its siblings in {subclasses}. First, the refactoring changes the visibility of {operation} – only if it is not *private* – to *protected* or *private*, depending whether it has subclasses. Finally, *non-private* siblings of {class} in {subclasses} become *protected*.

Algorithm:

1. if visibility of operation is not *private* then – (if {operation} has no subclasses then *{operation}.changeVisibility(private)* else *{operation}.changeVisibility(protected)*)
2. for each sibling operation of {operation} in {subclasses} – if visibility of {sibling_operation} is not *private* then *{sibling_operation}.changeVisibility(protected)*

Formal specification

```
context Operation::HideOperation()

pre : not isActive --1
and session.getAllCallExpr()->forAll(ce: CallExpr | Set{self}-
>union(self.getSiblingOperationsSub())->exists(ce.called) implies ce.to-
>isEmpty) --2&3

post: if namespace.allSubclasses->isEmpty or visibility@pre = #VkPrivate then
visibility = #VkPrivate else visibility = #VkProtected endif --1 and
self.getSiblingOperationsSub()->forAll(o: Operation | o.visibility@pre <>
#VkPrivate implies o.visibility = #VkProtected) --2
```

7.3.4 Add Parameter to Operation (Passive)

Informal specification

Origin: Add Parameter [Fowler *et al.* 1999]
Areas: ESPC
Trigger-element: PassiveClass::Operation

Definitions:

1. parameter – a parameter that is to be added
2. name – the name of {parameter}
3. type – the type of {parameter}
4. operation [T] – an operation to which {parameter} is added
5. class – a class in which {operation} is defined

Aim: Add {parameter} to {operation}

Reasons: {operation} needs more information from its callers

Description

{parameter} is added to {operation} and to all its siblings in the inheritance hierarchy of {class}.

Preconditions:

1. {class} is passive.
2. {name} adheres to TAU naming rules.
3. {operation} does not have a parameter called {name}.
4. There is no local variable called {name} in the bodies of {operation} and its siblings in the inheritance hierarchy of {class}.
5. There is no attribute called {name} in the inheritance hierarchy of {class}.
6. No operation with the signature implied by adding {parameter} to {operation} exists in the inheritance hierarchy of {class}.

Postconditions:

1. {parameter} exists – at the last position – in the signatures of {operation} and its siblings.
2. {operation} and its siblings are invoked with {parameter} having a default value.

Mechanics

After creation of {parameter}, it is added to {operation} and to all its siblings in the inheritance hierarchy of {class}. Next, in each invocation of {operation} and all its siblings, a new actual argument with a default value (NULL) is appended.

Algorithm:

1. *{operation}.appendParameter({name},{type})*
2. for each sibling operation of {operation} – *{sibling_operation}.appendParameter({name},{type})*
3. for each invocation of {operation} and its siblings – *{operation_invocation}.addDefaultValueForParameter({name})*

Formal specification

```
context Operation::AddParameter(p: Parameter)

pre : not namespace.isActive --1
and p.name.isValidName() --2
and not parameter->exists(pa: Parameter | pa.name = p.name) --3
and not Set{self}->union(self.getSiblingOperationsSub())-
>union(self.getSiblingOperationsSup())->exists(o: Operation |
o.hasLocalVariableNamed(p.name)) --4
and not namespace.getAllClassesFromInheritanceHierarchy().attribute->exists(pa:
Parameter | pa.name = p.name) --5
```

```
and not namespace.getAllClassesFromInheritanceHierarchy().behavioralFeature-
>exists(o: Operation | o.name = self.name and
o.hasTheSameParameterList(self.parameter->append(p))) --6

post: let allOp = Set{self}->union(self.getSiblingOperationsSub())-
>union(self.getSiblingOperationsSup()) in
allOp->forAll(o: Operation | let lp = o.parameter->last() in lp.name = p.name
and lp.type = p.type) --1
and session.getAllCallExpr()->forAll(ce: CallExpr | allOp->exists(o: Operation
| o = ce.called) implies (let arg = ce.argument->last() in
arg.oclIsTypeOf(Ident) and arg.definition.oclIsTypeOf(Literal) and
arg.definition.name = 'NULL')) --2
```

7.4 Trigger-element – Parameter of Operation

This section contains specifications of following refactorings:

1. Remove Parameter from Operation (Passive)

7.4.1 Remove Parameter from Operation (Passive)

Informal specification

Origin: Remove Parameter [Fowler *et al.* 1999]
Areas: ESPC
Trigger-element: PassiveClass::Operation::Parameter

Definitions:

1. parameter [T] – a parameter that is to be removed
2. operation – an operation in which {parameter} is defined
3. class - a class in which {operation} is defined

Aim: Remove {parameter} from {operation}

Reasons: {parameter} is no longer needed by the implementation of {operation}

Description

{parameter} is removed from {operation} and from all its siblings in the inheritance hierarchy of {class}.

Preconditions:

1. {class} is passive.
2. {parameter} is not used in the bodies of {operation} and its siblings from the inheritance hierarchy of {class}.
3. No operation with the signature implied by removing {parameter} from {operation} exists in the inheritance hierarchy of {class}.

Postconditions:

1. {parameter} does not exist in signatures of {operation} and its siblings.
2. {operation} and its siblings are invoked without {parameter}.

Mechanics

First, {parameter} is removed from {operation}, and next, its sibling parameters are deleted from all siblings of {operation}.

Algorithm:

1. *{parameter}.removeParameter()*
2. for each sibling parameter of {parameter} – *{sibling_parameter}.removeParameter()*

Formal specification

```
context Parameter::RemoveParameter()

pre : not eventClass.isActive --1
and Set{self.eventClass}->union(self.eventClass.getSiblingOperationsSub())-
>union(self.eventClass.getSiblingOperationsSup())->forAll(o: Operation | not
o.isParameterUsed(self.name)) --2
and not
eventClass.namespace.getAllClassesFromInheritanceHierarchy().behavioralFeature-
>exists(o: Operation | o.name = self.eventClass.name and let pList =
self.eventClass.parameter in o.hasTheSameParameterList(pList->excluding(pList-
>last()))) --3

post: let allOp = Set{self.eventClass}-
>union(self.eventClass.getSiblingOperationsSub())-
>union(self.eventClass.getSiblingOperationsSup()) in
not allOp->exists(o: Operation | o.parameter->exists(p: Parameter | p.name =
self.name and p.type = self.type)) --1
and session.getAllCallExpr()->forAll(ce: CallExpr | allOp->exists(ce.called)
implies ce.argument->asSequence() = ce.argument@pre->asSequence()-
>excluding(ce.argument@pre->asSequence()->last())) --2

context Operation::isParameterUsed(pName: String): Boolean
body: inlineMethod.getAllIdent()->exists(i: Ident |
i.definition.oclIsTypeOf(Parameter) and i.definition.name = pName)
```

7.5 Common OCL Queries

```
context String::consistsOfAllowedCharacters(): Boolean
/* the query returns true if the context element consists of characters allowed
in names of model elements (refer to [Telelogic 2004] - Names) */

context Session::getAllCallExpr(): Set(CallExpr)
/* the query returns a set containing all instances of CallExpr in the model */

context Class::getAllClassesFromInheritanceHierarchy(): Set(Class)
body: self->union(self.allSupertypes)->union(self.allSubtypes)

context OperationBody::getAllIdent(): Set(Ident)
/* the query returns all instances of Ident used in all expressions in the
context operation body */

context CompoundAction::getAllSimpleActions(): Set(Action)
body: self->iterate(a: Action; resultActions: Set(Action) = {} |
if a.oclIsTypeOf(CompoundAction) then
resultActions->union(a.getAllSimpleActions())
else resultActions->union(a) endif)

context Operation::getSiblingOperationsSub(): Set(Operation)
body: namespace.allSubclasses.behavioralFeature->select(o: Operation | o.name =
self.name and o.hasTheSameParameters(self))

context Operation::getSiblingOperationsSup(): Set(Operation)
body: namespace.allSuperclasses.behavioralFeature->select(o: Operation | o.name
= self.name and o.hasTheSameParameters(self))

context Operation::hasLocalVariableNamed(n: String): Boolean
body: inlineMethod->notEmpyt and inlineMethod.action.getAllSimpleActions()-
>exists(a: DefAction | a.definition.name = n)

context Operation::hasTheSameParameterList(p: Sequence(Parameter)): Boolean
body: parameter->reject(direction = #DkReturn)->collect(type) = p-
>reject(direction = #DkReturn)->collect(type)
```

```
context Operation::hasTheSameParameters(ops: Set(Operation)): Boolean
body: ops->exists(o: Operation | o.hasTheSameParameterList(self.parameter))

context String::isValidName(): Boolean
body: not isReservedWord() and consistsOfAllowedCharacters()

context String::isReservedWord(): Boolean
/* the query returns true if the context element is one of the reserved words
(refer to [Telelogic 2004] - Names) */
```

8 IMPLEMENTATION OF REFACTORINGS IN TAU

The goal of this chapter is to show how refactorings specified in the previous Section can be implemented in TAU. It is structured as follows: Section 8.1 provides basic information on model access in TAU; Section 8.2 describes implementation of an exemplary refactoring – Remove Middle Man – that has been performed within the frames of the thesis. The full source code of the application can be found on the attached CD. Finally, Section 8.3 explains how refactorings that are not driven by bad smells can be triggered on presentation elements.

8.1 Model Access in TAU

A TAU model can be accessed either using TCL Script or via a COM API from an application written in any COM-enabled programming language. The latter solution offers higher execution performance, but on the other hand, it requires more development effort. The COM API is primarily intended to be used by applications running in their own memory space, referred to as non-interactive clients. They work on their own private copy of a model, and the information exchange with other applications is therefore typically at file level. However, the COM API can also be used by interactive clients that access the model loaded by TAU.

A non-interactive client accesses the COM API by creating an instance of the only one exposed COM class `TTD_ModelAccess`, which implements `ITtdModelAccess` interface. An interactive client must implement `ITtdInteractiveClient` interface. This interface contains a method `OnExecute` which is called by TAU when the interactive client it to execute. The first argument of `OnExecute` method is a pointer to an `ITtdInteractiveServer` interface, which represents the application which acts as the server for the interactive client. The second argument of `OnExecute` method is an `ITtdEntities` collection of entities.

Unfortunately, it is currently not possible to develop the entire add-in using only a COM client – TAU requires at least a minimal TCL script to execute. In order to transfer execution to the COM client, this script can use a TCL API command `ExecuteCOMClient` (see UMLref.tcl on attached CD).

8.2 Implementation of Remove Middle Man

Taking into account probable performance and usability requirements imposed on a refactoring add-in, the access via the COM API has been chosen – the exemplary refactoring is implemented as an interactive client – called *UMLref* – written in Borland Delphi's Object Pascal.

The structure of *UMLref* can be overviewed in **Figure 8.1**. The implementation consists of seven packages representing Delphi units. Each package contains one class named after its container. TMiddleMan is responsible for detecting middle men in models, TMiddleManUI is a user interface class enabling selection of a middle man and triggering the transformation, and TRemoveMiddleMan implements the refactoring. TClassMV and TInvocationReplacerMV are two classes which presence is enforced by the use of the COM API. The former has the ability to traverse a model and to find all potential middle men, and the latter searches for all invocations of

delegating operations. They both realize `ITtdMetaVisitCallback` interface, containing `OnVisitedEntity` method called for each entity that is visited during the model traversal. Two last classes, namely TChecker and TTransformer, contain operations used mainly in checking pre- and postconditions, and transforming models, respectively. These two classes are the only ones that are intended to be reused in implementations of other refactorings.

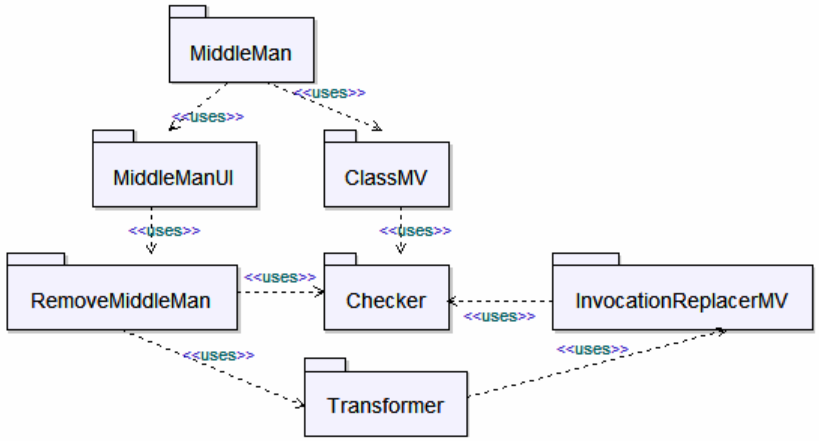

Figure 8.1. Package structure of the implementation of Remove Middle Man

Each bad smell driven refactoring is performed in two phases – in the case of Remove Middle Man, in the former, a model is traversed in the search for classes that are suspected to be middle men, and in the latter, a user triggers the transformation on a chosen pair consisting of a class and one of its delegate attributes. An interaction illustrating the first phase is shown in **Figure 8.2**. In the source code, `findMiddleMen` operation is realized by `OnVisitedEntity` operation.

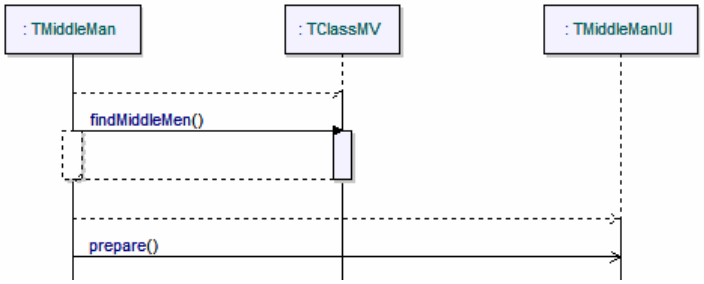

Figure 8.2. A sequence diagram illustrating detection of middle men

An overall refactoring algorithm for the second phase can be defined as follows:

```
if preconditionsFulfilled() then
  begin
    transform();
    if not postconditionsFulfilled() then
      rollback();
  end;
```

Its realization can be overviewed in a sequence diagram in **Figure 8.3**.

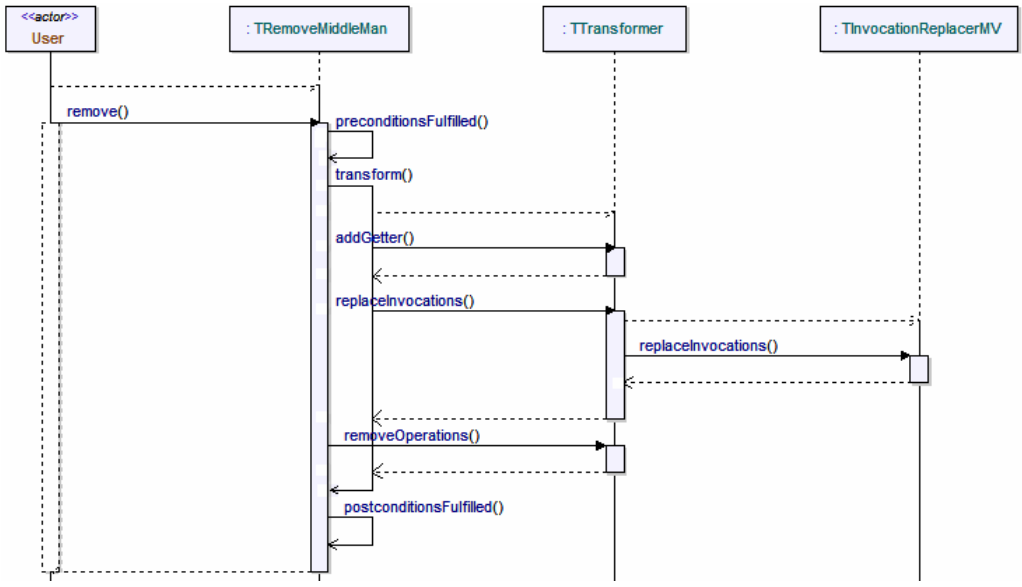

Figure 8.3. A sequence diagram illustrating a successful removal of a middle man

Although the current version (2.4) of TAU does not provide support for OCL on the metamodel level, transition from the formal specification of both queries detecting bad smells as well as pre- and postconditions to their implementation is quite straightforward. It results from the fact that interfaces exposed by TAU COM API, realized by classes representing metaclasses in the implementation of TAU Object Model, enable – just like OCL – a metamodel-based navigation through models[40]. Therefore, for instance two first preconditions and the first and the third postcondition of Remove Middle Man boil down to the following statements:

```
Pre  1) result:=dAttr.GetOwner()=mMan
Pre  2) result:=dAttr.GetEntity('type',1).GetMetaClassName='Class'
Post 1) result:=checker.hasGetter(mMan,dAttr)
Post 3) result:=not checker.hasDelegatingOps(mMan,dAttr)
```

In the case of the transforming part, it is noteworthy that the three basic operations identified in the mechanics section of the informal specification of Remove Middle Man, i.e. *addGetter*, *replaceInvocation*, and *removeOperation*, have their correspondents in the implementation (see **Figure 8.3**). For instance, *replaceInvocations* procedure calls *replaceInvocation* for each instance of *CallExpr* that calls a delegating operation of a suspected class. The implementation of *replaceInvocation* (see below) transforms the model in accordance with **Figure 7.7**.

```
1  procedure TInvocationReplacerMV.replaceInvocation
2            (callExpr, accessor, op: ITtdEntity);
3  (…)
4    toPre:=callExpr.GetEntity('to',1);
5    model:=callExpr;
6    while not model.IsKindOf('Session') do
7      model:=model.GetOwner;
8    newCallExpr:=(model as ITtdModel).New('CallExpr');
9    newCallExpr.SetEntity('called',accessor,1);
10   newCallExpr.SetEntity('to',toPre,1);
11   callExpr.SetEntity('called',op,1);
12   callExpr.SetEntity('to',newCallExpr,1);
```

First (4), an identifier, pointing an element on which a delegating operation is invoked, is stored in a temporary variable `toPre`. Next (5-7), the instance of `Session`

[40] This way, changes to TAU Object Model do not propagate to the COM API, but on the other hand, this solution requires thorough knowledge and comprehension of TAU's metamodel.

metaclass is found, to be able to create (8) a new instance of `CallExpr`[41]. Subsequently, the most important attributes (`called` and `to`) of the created call expression are set to point adequate elements (9-10). Finally, the initial invocation is adjusted to call an appropriate delegated operation via the new call expression (11-12).

8.3 Triggering Refactorings

In the case of refactorings that are not driven by bad smells, there is a need to enable triggering them on both model elements in the model view as well as on their representations in diagrams. In the latter case, a corresponding model element has to be determined and passed to the transformation. **Figure 8.4** shows relation between *ClassSymbol*, *GeneralizationLine*, and *AssociationLine* presentation metaclasses and their model counterparts pointed by *modelElement* attribute and represented by corresponding (indirect) subclasses of *ModelElement* – *Class*, *Generalization*, and *Association*.

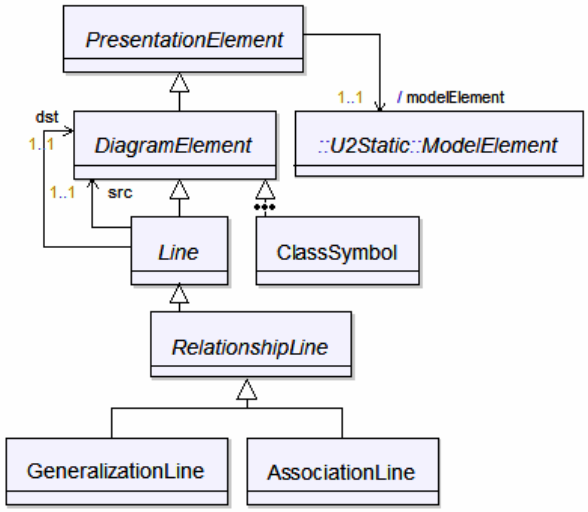

Figure 8.4. Relation between presentation elements and model elements

It is noteworthy that *modelElement* of *AssociationLine* is always not *Association* but *Attribute*. Additionally, in the case of *GeneralizationLine* and *AssociationLine*, two attributes – *dst* and *src* – point *ClassSymbols* of related classes. The source for *Generalization* is a superclass and its destination – a subclass.

Having established relation between presentation and model elements, one can trigger refactorings from TCL scripts in the following way:

```
set sel [std::GetSelection]
if { [u2::GetMetaClassName $sel] == "ClassSymbol" } {
            set sel [u2::GetEntity $sel "ModelElement"]
}
std::ExecuteCOMClient "UMLref.RenameClass" $sel
```

where GetSelection returns selected in TAU model/presentation element, and ExecuteCOMClient invokes an interactive COM client that performs – in this case – Rename Class refactoring.

[41] In this particular case, the use of `Create` method defined in `ITtdEntity` interface, instead of `New`, fails – this is probably due to a bug in the COM API.

9 CONCLUSIONS & FUTURE WORK

This is the last chapter of the thesis. It concludes it (Section 9.1) and points out possible directions of a future work (Section 9.2).

9.1 Conclusions

All objectives of the thesis have been successfully accomplished. Literature survey on software refactoring (Section 2), UML model refactoring (Section 3), and executable modelling with UML (Section 4) has been performed. An initial catalogue of executable UML model refactorings has been created (Section 7), and the transformations have been formalized with the use of OCL. Finally, an exemplary refactoring from the catalogue has been implemented in TAU (Section 8).

The majority of previous studies on UML model refactoring presented in Section 3 concern refactoring of non-executable UML models, in which operation bodies are treated as *protected areas*. The main difference between transformations used in these approaches and refactorings of *executable* models relies on the fact that the latter ones have to take into account and to update not only structural but also behavioural aspects of transformed models. The key challenges in the area of refactoring of executable models that base on UML 2.0 result mainly from the necessity to consider following new features of the language: (1) cross integration of structure and behaviour, (2) support for component-based development via composite structures, and (3) integration of action semantics with behavioural constructs.

With the intention of enabling a systematic approach to the mentioned issues, in this thesis a notion of *refactoring trigger-element* has been introduced. Next, trigger-elements of all Fowler's code refactorings [Fowler *et al.* 1999] have been determined. Subsequently, a mapping between all these trigger-elements and their equivalents in the UML 2.0 metamodel and TAU Object Model have been established, indicating that all code refactorings can be applied to executable UML models. The thesis elaborates categorization of model refactorings in the form of *refactoring areas* based on the notion of trigger-elements. Exemplary transformations from each area are presented, and the overall ideas are illustrated on a study executable UML model built in TAU. The identified refactoring areas are specific for TAU executable models, but one may expect that similar ones could be distinguished in e.g. I-Logix Rhapsody models. Basing on the initial research, a systematic approach to specification of both executable UML model refactorings as well as associated bad smells has been elaborated. Using a proposed specification template, twelve refactorings have been specified, and one of them – Remove Middle Man – implemented in TAU. It is worthy noting that both specifications as well as implementations of the transformations are specific for Telelogic TAU, i.e. they are not straightforwardly portable to other UML CASE tools. This results mainly from the fact that TAU Object Model substantially differs from UML 2.0 metamodel.

The issue of refactoring executable UML models, introduced by Sunyé *et al.* [2002], is addressed by Kazato *et al.* [2004] (see Section 3.7) in a paper, which seems to be the only one on the topic published so far. Although their approach has a significant research value, its practical application seems rather inconvenient. First, it is not dedicated to UML CASE tools, but to graph transformation systems. This implies that each attempt to refactor a design model requires exporting, transforming,

and importing it back again[42]. In the current prototype tool, each refactoring is performed in accordance with the following scenario:

1. Edit the model in the transformation system;
2. Create *refactoring node* and set its "name" attribute to the name of the desired refactoring;
3. Connect the node to the model element relevant to the transformation by using "target" edge;
4. Launch the transformation.

Second, the design models refactored by Kazato *et al*. are based on an already depreciated UML 1.5 instead of being compliant with the completely rebuilt state-of-the-art UML 2.0. Moreover, refactored are only stateless models, which have more in common with programs written in modern object-oriented programming languages than with state-oriented executable models.

In comparison with the work performed by Kazato *et al.*, this thesis focuses on refactoring more complex, state-based TAU executable models. What is also important from the point of view of the practitioner building UML models with a professional tool such as TAU, the refactoring transformations and detection of related bad smells are programmed in the tool using its metamodel-based COM API instead of implementation of the transformations in a foreign environment such as a graph-rewriting tool.

The key problems encountered during the work are connected mainly with TAU Object Model, which is quite complex (ca. 200 metaclasses), not compliant with the UML 2.0 metamodel, and supplied only in the form of a model that can be opened and browsed in TAU [Telelogic 2005]. Moreover, the part that can be used while building executable UML models is not strictly defined.

9.2 Future Work

Several topics can be pointed out as a possible future work based on the approach presented in this thesis. These are as follows:

1. **Identification and definition of refactorings and associated bad smells specific for executable UML models.** By specific understood are the ones that cannot be applied/do not occur in programming language code. These are mainly refactorings triggered on elements in ISAC and LCAC areas. Two examples of them, namely Extract Port and Group States, are given in Section 6.8.1 and 6.8.2, respectively.

2. **Development of a professional industrial refactoring add-in to TAU.** This add-in would (1) support users in detecting bad smells and (2) suggest appropriate refactorings to remove them. Moreover, it would enable to trigger user-defined transformations.

3. **Automation of generating implementations of refactorings from their formal specifications.** This topic bases on an observation that implementation of both queries as well as pre- and postconditions in TAU COM API, from OCL specifications, is straightforward.

[42] Assuming that the graph transformation system is not a part of the UML CASE tool.

4. **Refactoring of executable models in a tool fully compliant with UML 2.0.** Refactorings specified for such a tool would be portable across all tools compliant with UML 2.0 – as soon as they appear on the market.

5. **Specification of refactorings with the use of action semantics.** This topic bases on an observation that action semantics can be used also to transform UML models [Sunyé *et al.* 2002; Varro & Pataricza 2003].

6. **Discovering refactoring preconditions in a systematic manner.** The most problematic step during specifying a refactoring seems to be determination of its preconditions. This results from the fact that is very difficult to foresee all situations is which the refactoring will not be behaviour preserving.

7. **An experiment with human subjects, evaluating:**
    a. influence of automation of executable UML model refactoring on productivity of software developers,
    b. the effect of the refactoring on maintainability of executable UML models.

8. **Construction of a catalogue of executable UML model refactorings.** The refactorings should be specified according to the template enhanced by among others sections concerning (1) classification of refactorings, (2) related refactorings, and (3) consequences of refactorings.

10 REFERENCES

1. Alur, D., Crupi, J. and Malks, D. (2001) *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall.
2. Ambler, S.W. (2003) 'Be Realistic About the UML: It's Simply Not Sufficient', available from Internet <http://www.agilemodeling.com/essays/realisticUML.htm> (5 April 2005).
3. Astels, D. (2002) 'Refactoring with UML', in *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, Alghero, 26 – 29 May 2002, 67-70.
4. Beck, K. (2000) *Extreme Programming Explained: Embrace Change*, Addison Wesley.
5. Beck, K. and Fowler, M. (2001) *Planning Extreme Programming*, Addison Wesley.
6. Bennett, K.H. and Rajlich, V.T. (2000) 'Software maintenance and evolution: a roadmap', in *Proceedings of the 22nd International Conference on Software Engineering*, 75-87.
7. Björkander, M. (2000) 'Graphical programming using UML and SDL', *Computer*, **33**(12), 30-35.
8. Björkander, M. and Kobryn, C. (2003) 'Architecting Systems with UML 2.0', *IEEE Software*, **20**(4), 57-61.
9. Boger, M., Sturm, T. and Fragemann, P. (2003) 'Refactoring Browser for UML', *Lecture Notes in Computer Science*, **2591**, 366-377.
10. Bock, C. (2003) 'UML without Pictures', *IEEE Software*, **20**(5), 33-35.
11. Booch G. (2002) 'The Future of Software Technology', *CHIPS Magazine*, **20**(4).
12. Bosch, J. (2000) *Design and use of software architectures*, Addison-Wesley.
13. Bottoni, P., Parisi-Presicce, F. and Taentzer, G. (2003) 'Coordinated Distributed Diagram Transformation for Software Evolution', *Electronic Notes in Theoretical Computer Science*, **72**(4), 1-12.
14. Boyd, G. (2003) 'Executable UML: Diagrams for the Future', available from Internet <http://www.devx.com/enterprise/Article/10717> (30 December 2004).
15. Chapin, N., Hale, J.E., Md. Khan, K., Ramil, J.F. and Tan, W.-G. (2001) 'Types of software evolution and software maintenance', *Journal of Software Maintenance and Evolution: Research and Practice*, **13**, 3-30.
16. Chikofsky, E.J. and Cross, J.H.II (1990) 'Reverse engineering and design recovery: a taxonomy', *IEEE Software*, **7**(1), 13-17.
17. Correa, A.L. and Werner, C.M.L. (2004) 'Applying Refactoring Techniques to UML/OCL Models', in *International Conference on the Unified Modeling Language (UML'04)*, Lisbon, October 2004, 173-187.
18. Dawson, C.W. (2000) *The essence of computing projects : a student's guide*, Pearson Education Limited.
19. Demeyer, S. (2002) 'Maintainability versus Performance: What's the Effect of Introducing Polymorphism', Technical Report, University of Antwerp, Belgium.
20. Demeyer, S., Ducasse, S. and Tichelaar, S. (1999) 'Why Unified is not Universal: UML Shortcomings for Coping with Round-trip Engineering', in *Proceedings of 2nd International Conference UML'99 – Unified Modeling Language – Beyond the Standard* (Lecture Notes in Computer Science **1723**), 630-44.
21. Du Bois, B., Van Gorp, P., Amsel. A., Van Eetvelde, N., Stenten, H., Demeyer, S. and Mens, T. (2004) 'A discussion of refactoring in research and practice', Technical Report (number 2004-03), University of Antwerp, Belgium.

22. Feng, T. (2003) 'Action Semantics for an Executable UML', available from Internet <http://moncs.cs.mcgill.ca/people/tfeng/docs/as/> (1 January 2005).

23. Fowler, M. (1999) 'Alpha list of Refactorings', available from Internet <http://www.refactoring.com/catalog/index.html> (4 October 2004).

24. Fowler, M. (2003) *UML Distilled*, 3rd edition, Addison Wesley.

25. Fowler, M. (2004) 'RefactoringMalapropism', available from Internet <http://martinfowler.com/bliki/RefactoringMalapropism.html> (2 October 2004).

26. Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999) *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

27. France, R. and Bieman, J.M. (2001) 'Multi-View Software Evolution: A UML-based Framework for Evolving Object-Oriented Software', in *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, Florence, 6 – 10 November 2001, 386-395.

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

29. Hightower, J. (1996) 'On the Evolution of Programming', *Colorado Engineer Magazine*, vol. Fall-Winter.

30. Ho, W.-M.., Jézéquel, J.-M.., Le Guennec, A. and Pennaneac'h, F. (1999) 'UMLAUT: an extendible UML transformation framework', in *Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, 12 – 15 October 1999, Florida, 275-278.

31. Ho, W.-M.., Pennaneac'h, F. and Plouzeau, N. (2000) 'UMLAUT: a framework for weaving UML-based aspect-oriented designs', in *Proceedings of 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 33)*, 5 – 8 June 2000, St. Malo, 324-334.

32. IEEE (1990) *IEEE standard glossary of software engineering terminology*, IEEE Std 610.12-1999.

33. Kataoka, Y., Imai, T., Andou, H. and Fukaya, T. (2002) 'A quantitative evaluation of maintainability enhancement by refactoring', in *Proceedings International Conference on Software Maintenance*, 576-585.

34. Kazato, H., Takaishi, M., Kobayashi, T. and Saeki. M. (2004) 'Formalizing Refactoring by Using Graph Transformation', *IEICE Transactions on Information and Systems*, **E87-D**(4), 89-92.

35. Kerievsky, J. (2004) *Refactoring to Patterns*, Addison-Wesley Professional.

36. Kobryn, C. (2004) 'UML 3.0 and the future of modeling', *Software and System Modeling*, **3**(1), 4-8.

37. Kobryn, C. and Samuelsson, E. (2003) 'Driving Architectures with UML 2.0 – The TAU Generation2 Approach to Model Driven Architecture', White Paper, available from Internet <http://www.telelogic.com/resources/get_file.cfm?id=3537&filetype=Other> (27 April 2005).

38. LC (2004) 'Function Point Counting', Longstreet Consulting Inc., available from Internet <http://www.ifpug.com/fpc.htm> (3 January 2005).

39. LC (2004a) 'Software Productivity Since 1970', Longstreet Consulting Inc., available from Internet <http://www.ifpug.com/Articles/history.htm> (3 January 2005).

40. Leblanc, P. (2004) 'UML 2.0 Action Semantics and Telelogic TAU/Architect and TAU/Developer Action Language', White Paper, available from Internet <http://whitepapers.zdnet.co.uk/0,39025945,60094585p-39000629q,00.htm> (27 April 2005).

41. Lehman, M.M. (1980) 'On understanding laws, evolution and conservation in the large program life cycle', *Journal of Systems and Software*, **1**(3), 213-221.

42. Mäntylä, M. (2003) 'Bad Smells in Software – a Taxonomy and an Empirical Study', MSc thesis, Department of Computer Science and Engineering, Helsinki University of Technology.

43. Mäntylä, M. (2004) 'A Taxonomy for Bad Code Smells', available from Internet <http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm> (8 December 2004).

44. Massoni, T. (2003) 'Introducing Refactoring to Heavyweight Software Processes', Technical Report, CIn-UFPE, Brasil.

45. Mellor, S.J. and Balcer, M.J. (2002) *Executable UML: A Foundation for Model Driven Architecture*, Addison-Wesley.

46. Mellor, S.J., Scott, K., Uhl, A. and Weise, D. (2004) *MDA Distilled*, Addison-Wesley Professional.

47. Mens, T. and Tourwé, T. (2004) 'A Survey of Software Refactoring', *IEEE Transactions on Software Engineering*, **30**(2), 126-139.

48. Mens, T. and Van Deursen, A. (2003) 'Refactoring: Emerging Trends and Open Problems', in *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, British Columbia, 13 November 2003.

49. Mens, T., Demeyer, S., Du Bois, B., Stenten, H. and Van Gorp, P. (2002) 'Refactoring: Current research and future trends', *Third Workshop on Language Descriptions, Tools and Applications (LDTA)*, Warsaw, 6 April 2003.

50. Mens, T., Van Eetvelde, N., Janssens, D. and Demeyer, S. (2005) 'Formalising Refactorings with Graph Transformations', *Journal of Software Maintenance and Software Evolution*, scheduled for publication in July/August Issue.

51. Niemann, S. (2004) 'Executable Systems Design with UML 2.0', available from Internet <http://www.omg.org/news/whitepapers/Executable_System_Design_UML.pdf> (15 April 2005).

52. Ó Cinnéide, M. (2000) 'Automated Application of Design Patterns: A Refactoring Approach', PhD thesis, Department of Computer Science, Trinity College, University of Dublin.

53. OMG (2002) *Unified Modeling Language Specification Version 1.4.2*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/doc?formal/04-07-02.pdf> (29 November 2004).

54. OMG (2002a) *UML 1.4 with Action Semantics, Final Adopted Specification*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/do_doc?ptc/02-01-09.pdf> (7 April 2005)

55. OMG (2003) *UML 2.0 Infrastructure Final Adopted specification*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-15.pdf> (1 May 2005).

56. OMG (2003a) *UML 2.0 Diagram Interchange Final Adopted specification*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-01.pdf> (1 May 2005).

57. OMG (2003b) *UML 2.0 OCL Final Adopted specification*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf> (1 May 2005).

58. OMG (2004) *UML 2.0 Superstructure Revised Final Adopted specification (convenience document)*, Object Management Group, available from Internet <http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-02.pdf> (8 April 2004).

59. Opdyke, W.F. (1992) 'Refactoring Object-Oriented Frameworks', PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

60. Parnas, D.L. (1994) 'Software Aging', in *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, 279-287.

61. Pender, T. (2003) *UML Bible*, Wiley.

62. Pigoski, T.M. (1997) *Practical Software Maintenance – Best Practices for Managing Your Software Investment*, John Wiley & Sons.

63. Pollet, D., Vojtisek, D. and Jézéquel, J.-M. (2002) 'OCL as a Core UML Transformation Language', WITUML Position Paper at *16th European Conference on Object-Oriented Programming*, Málaga, 10-14 June 2002.

64. Porres, I. (2003) 'Model Refactorings as Rule-Based Update Transformations', Technical Report Series No. 525, Turku Center for Computer Science, Finland.

65. Riehle, D., Fraleigh, S., Bucka-Lasses, D. and Omorogbe, N. (2001) 'The architecture of a UML virtual machine', *SIGPLAN Notices*, **36**(11), 327-341.

66. Roberts, D.B. (1999) 'Practical Analysis for Refactoring', PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

67. Rumpe, B. (2002) 'Executable modelling with UML: a vision or a nightmare?', *Issues and Trends of Information Technology Management in Contemporary Organizations. 2002 Information Resources Management Association International Conference*, **1**(1), 697-701.

68. Schattkowsky, T. and Müller, W. (2004) 'Model-Based Design of Embedded Systems', in *Proceedings of Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing,* 121-128.

69. Schattkowsky, T. and Müller, W. (2004a) 'Model-Based Specification and Execution of Embedded Real-Time Systems', in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, 1392-1393.

70. Sendall, S. and Kozaczyński, W. (2003) 'Model transformation: the heart and soul of model-driven software development', *IEEE Software*, **20**(5), 42-45.

71. Sendall, S., Hauser, R., Koehler, J., Küster, J. and Wahler, M. (2004) 'Understanding Model Transformation by Classification and Formalization', in *Proceedings of Workshop on Software Transformation Systems* (part of 3rd International Conference on Generative Programming and Component Engineering), Vancouver, 24 October 2004.

72. Selic, B.V. (2004) 'On the Semantic Foundations of Standard UML 2.0', *Lecture Notes in Computer Science*, **3185**, 181-199.

73. Staroń, M. and Kuźniarz, L. (2004) 'Implementing UML Model Transformations for MDA', *11th Nordic Workshop on Programming and Software Development and Tools*, Turku, Finland.

74. Starr, L. (2002a) *Executable UML. How to build class models*, Prentice Hall.

75. Starr, L. (2002b) 'Executable UML Metamodel', available from Internet <http://www.modelint.com/download.html> (11 October 2004).

76. Sunyé, G., Pollet, D., Le Traon, Y. and Jézéquel, J.-M. (2001) 'Refactoring UML Models', *Lecture Notes in Computer Science*, **2185**, 134-148.

77. Sunyé, G., Le Guennec, A. and Jezequel, J.M. (2002) 'Using UML Action Semantics for Model Execution and Transformation', *Information Systems,* **27**(6), 445-457.

78. Svahnberg, M. (2003) 'Supporting Software Architecture Evolution. Architecture Selection and Variability', PhD thesis, Department of Software Engineering and Computer Science, Blekinge Institute of Technology.

79. Swanson, E.B. (1976) 'The Dimensions of Maintenance', in *Proceedings of the 16th International Conference on Software Engineering*, *IEEE Computer Society*, 492-497.

80. Telelogic (2004) 'Telelogic TAU 2.4 Help', a help file attached to TAU.

81. Telelogic (2005) 'TAU Object Model', available from Internet for registered TAU users <https://support.telelogic.com/en/tau/addins/index.cfm?contentid=8046&fieldid=3906> (20 April 2004).

82. Tichelaar, S. (2001) 'Modeling Object-Oriented Software for Reverse Engineering and Refactoring', PhD thesis, Faculty of Science, University of Bern.

83. Tichelaar, S., Ducasse, S., Demeyer, S. and Nierstrasz, O. (2000) 'A Meta-model for Language-Independent Refactoring', in *Proceedings International Symposium on Principles of Software Evolution*, Kanazawa, 1 – 2 November 2000, 154-164.

84. Van Gorp, P., Stenten, P., Mens, T. and Demeyer, S. (2003a) 'Enabling and using the UML for model driven refactoring', in *Proceedings of the 4th International Workshop on Object-Oriented Reengineering (WOOR)*, Darmstadt, 21 July 2003.

85. Van Gorp, P., Stenten, P., Mens, T. and Demeyer, S. (2003b) 'Towards automating source-consistent UML refactorings', in *Proceedings of the 6th International Conference on UML – The Unified Modeling Language*, San Francisco, 20 – 24 October 2003.

86. Van Gorp, P., Van Eetvelde, N. and Janssens, D. (2003c) 'Generating Refactoring Implementations from Platform Independent Metamodel Transformations', in *Proceedings International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI 2003)*, Luxembourg, 27 – 28 November 2003.

87. Van Gurp, J. and Bosch, J. (2002) 'Design Erosion: Problems & Causes', *Journal of Systems & Software*, **61**(2), 105-119.

88. Van Gurp, J., Smedinga, R. and Bosch, J. (2002) 'Architectural Design Support for Composition and Superimposition', in *Proceedings of IEEE HICCS 35*.

89. Varro, D. and Pataricza, A. (2003) 'UML actions semantics for model transformation systems', *Periodica Polytechnica Electrical Engineering*, **47**(3-4), 167-186.

90. Wake, W.C. (2003) *Refactoring Workbook*, Addison-Wesley.

91. Warmer, J. and Kleppe, A. (1999) *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Professional.

92. Zhang, J., Lin, Y. and Gray, J. (2004) 'Generic and Domain-Specific Model Refactoring using a Model Transformation Engine', *Model-driven Software Development – Research and Practice in Software Engineering*, accepted for publication in 2005.

APPENDIX A – FOWLER'S CODE REFACTORINGS

Columns:

- **Group** – name of a group to which refactorings belong
- **No.** – number of a refactoring
- **Refactoring** – name of a refactoring
- **Summary** – a problem and a solution statement [Fowler *et al.* 1999]

Group	No.	Refactoring	Summary
Composing Methods	1.	Extract Method	You have a code fragment that can be grouped together. *Turn the fragment into a method whose name explains the purpose of the method.*
	2.	Inline Method	A method's body is just as clear as its name. *Put the method's body into the body of its callers and remove the method.*
	3.	Inline Temp	You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings. *Replace all references to that temp with the expression.*
	4.	Replace Temp with Query	You are using a temporary variable to hold the result of an expression. *Extract the expression into a method. Replace all references to the temp with the method. The new method can then be used in other methods.*
	5.	Introduce Explaining Variable	You have a complicated expression. *Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.*
	6.	Split Temporary Variable	You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable. *Make a separate temporary variable for each assignment.*
	7.	Remove Assignments to Parameters	The code assigns to a parameter. *Use a temporary variable instead.*
	8.	Replace Method with Method Object	You have a long method that uses local variables in such a way that you can not apply Extract Method. *Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.*
	9.	Substitute Algorithm	You want to replace an algorithm with one that is clearer. *Replace the body of the method with the new algorithm.*
Moving Features Between Objects	10.	Move Method	A method is, or will be, using or used by more features of another class than the class on which it is defined. *Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.*
	11.	Move Field	A filed is, or will be, used by another class more than the class on which it is defined. *Create a new field in the target class, and change all its users.*
	12.	Extract Class	You have one class doing work that should be done by two. *Create a new class and move the relevant fields and methods from the old class into the new class.*
	13.	Inline Class	A class isn't doing very much. *Move all its features into another class and delete it.*
	14.	Hide Delegate	A client is calling a delegate class of an object. *Create methods on the server to hide the delegate.*
	15.	Remove Middle Man	A class is doing too much simple delegation. *Get the client to call the delegate directly.*
	16.	Introduce Foreign Method	A server class you are using needs an additional method, but you can't modify the class. *Create a method on the client class with an instance of the server class as its first argument.*
	17.	Introduce Local Extension	A server class you are using needs several additional methods, but you can't modify the class. *Create a new class that contains these extra methods. Make this extension class a subclass or a wrapper of the original.*

Group	No.	Refactoring	Summary
Organizing Data	18.	Self Encapsulate Field	You are accessing a field directly, but the coupling to the field is becoming awkward. *Create getting and setting methods for the field and use only those to access the field.*
	19.	Replace Data Value with Object	You have a data item that needs additional data or behavior. *Turn the data item into an object.*
	20.	Change Value to Reference	You have a class with many equal instances that you want to replace with a single object. *Turn the object into a reference object.*
	21.	Change Reference to Value	You have a reference object that is small, immutable, and awkward to manage. *Turn it into a value object.*
	22.	Replace Array with Object	You have an array in which certain elements mean different things. *Replace the array with an object that has a field for each element.*
	23.	Duplicate Observed Data	You have domain data available only in a GUI control, and domain methods need access. *Copy the data to a domain object. Set up an observer to synchronize the two pieces of data.*
	24.	Change Unidirectional Association to Bidirectional	You have two classes that need to use each other's features, but there is only a one-way link. *Add back pointers, and change modifiers to update both sets.*
	25.	Change Bidirectional Association to Unidirectional	You have a two-way association but one class no longer needs features from the other. *Drop the unneeded end of the association.*
	26.	Replace Magic Number with Symbolic Constant	You have a literal number with a particular meaning. *Create a constant, name it after the meaning, and replace the number with it.*
	27.	Encapsulate Field	There is a public field. *Make it private and provide accessors.*
	28.	Encapsulate Collection	A method returns a collection. *Make it return a read-only view and provide add/remove methods.*
	29.	Replace Record with Data Class	You need to interface with a record structure in a traditional programming environment. *Make a dumb data object for the record.*
	30.	Replace Type Code with Class	A class has a numeric type code that does not affect its behavior. *Replace the number with a new class.*
	31.	Replace Type Code with Subclasses	You have an immutable type code that affects the behavior of a class. *Replace the type code with subclasses.*
	32.	Replace Type Code with State/Strategy	You have a type code that affects the behavior of a class, but you can not use subclassing. *Replace the type code with a state object.*
	33.	Replace Subclass with Fields	You have subclasses that vary only in methods that return constant data. *Change the methods to superclass fields and eliminate the subclasses.*
Simplifying Conditional Expressions	34.	Decompose Conditional	You have a complicated conditional (if-then-else) statement. *Extract methods from the condition, then part, and else parts.*
	35.	Consolidate Conditional Expression	You have a sequence of conditional tests with the same result. *Combine them into a single conditional expression and extract it.*
	36.	Consolidate Duplicate Conditional Fragments	The same fragment of code is in all branches of a conditional expression. *Move it outside of the expression.*
	37.	Remove Control Flag	You have a variable that is acting as a control flag for a series of boolean expressions. *Use a break or return instead.*
	38.	Replace Nested Conditional with Guard Clauses	A method has conditional behavior that does not make clear the normal path of execution. *Use guard clauses for all the special cases.*
	39.	Replace Conditional with	You have a conditional that chooses different behavior depending on the type of an object.

Group	No.	Refactoring	Summary
		Polymorphism	*Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.*
	40.	Introduce Null Object	You have repeated checks for a null value. *Replace the null value with a null object.*
	41.	Introduce Assertion	A section of code assumes something about the state of the program. *Make the assumption explicit with an assertion.*
Making Method Calls Simpler	42.	Rename Method	The name of a method does not reveal its purpose. *Change the name of the method.*
	43.	Add Parameter	A method needs more information from its caller. *Add a parameter for an object that can pass on this information.*
	44.	Remove Parameter	A parameter is no longer used by the method body. *Remove it.*
	45.	Separate Query from Modifier	You have a method that returns a value but also changes the state of an object. *Create two methods, one for the query and one for the modification.*
	46.	Parameterize Method	Several methods do similar things but with different values contained in the method body. *Create one method that uses a parameter for the different values.*
	47.	Replace Parameter with Explicit Methods	You have a method that runs different code depending on the values of an enumerated parameter. *Create a separate method for each value of the parameter.*
	48.	Preserve Whole Object	You are getting several values from an object and passing these values as parameters in a method call. *Send the whole object instead.*
	49.	Replace Parameter with Method	An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method. *Remove the parameter and let the receiver invoke the method.*
	50.	Introduce Parameter Object	You have a group of parameters that naturally go together. *Replace them with an object.*
	51.	Remove Setting Method	A field should be set at creation time and never altered. *Remove any setting method for that field.*
	52.	Hide Method	A method is not used by any other class. *Make the method private.*
	53.	Replace Constructor with Factory Method	You want to do more than simple construction when you create an object. *Replace the constructor with a factory method.*
	54.	Encapsulate Downcast	A method returns an object that needs to be downcasted by its callers. *Move the downcast to within the method.*
	55.	Replace Error Code with Exception	A method returns a special code to indicate an error. *Throw an exception instead.*
	56.	Replace Exception with Test	You are throwing a checked exception on a condition the caller could have checked first. *Change the caller to make the test first.*
Dealing with Generalization	57.	Pull Up Field	Two subclasses have the same field. *Move the field to the superclass.*
	58.	Pull Up Method	You have methods with identical results on subclasses. *Move them to the superclass.*
	59.	Pull Up Constructor Body	You have constructors on subclasses with mostly identical bodies. *Create a superclass constructor; call this from the subclass methods.*
	60.	Push Down Method	Behavior on a superclass is relevant only for some if its subclasses. *Move it to those subclasses.*
	61.	Push Down Field	A field is used only by some subclasses. *Move the field to those subclasses.*
	62.	Extract Subclass	A class has features that are used only in some instances. *Create a subclass for that subset of features.*
	63.	Extract Superclass	You have two classes with similar features. *Create a superclass and move the common features to the superclass.*
	64.	Extract Interface	Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common. *Extract the subset into an interface.*
	65.	Collapse Hierarchy	A superclass and subclass are not very different. *Merge them together.*

Group	No.	Refactoring	Summary
	66.	Form Template Method	You have two methods in subclasses that perform similar steps in the same order, yet the steps are different. *Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.*
	67.	Replace Inheritance with Delegation	A subclass uses only part of a superclasses interface or does not want inherit data. *Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.*
	68.	Replace Delegation with Inheritance	You're using delegation and are often writing many simple delegations for the entire interface. *Make the delegating class a subclass of the delegate.*
Big Refactorings	69.	Tease Apart Inheritance	You have an inheritance hierarchy that is doing two jobs at once. *Create two hierarchies and use delegation to invoke one from the other.*
	70.	Convert Procedural Design to Objects	You have code written in a procedural style. *Turn the data records into objects, break up the behavior, and move the behavior to the objects.*
	71.	Separate Domain from Presentation	You have GUI classes that contain domain logic. *Separate the domain logic into separate domain classes.*
	72.	Extract Hierarchy	You have a class that is doing too much work, at least in part through many conditional statements. *Create a hierarchy of classes in which each subclass represents a special case.*

APPENDIX B – BAD SMELLS IN CODE

Columns:

- **Wake's Group** – name of a group to which bad smells belong, according to Wake's taxonomy [Wake 2003]
- **Wake's Subgroup** – name of a subgroup to which bad smells belong, according to Wake's taxonomy [ibid.]
- **Wake's Bad Smell** – name of a bad smell, according to Wake's taxonomy [ibid.]
- **Fowler's Bad Smell** – name of a corresponding bad smell from Fowler's taxonomy [Fowler *et al.* 1999]

Wake's Group	Wake's Subgroup	Wake's Bad Smell	Fowler's Bad Smell
Smells Within Classes	Measured Smells	Comments	Comments
		Long Method	Long Method
		Large Class	Large Class
		Long Parameter List	Long Parameter List
	Names	Type Embedded in Name (Including Hungarian)	n/a
		Uncommunicative Name	n/a
		Inconsistent Names	n/a
	Unnecessary Complexity	Dead Code	Speculative Generality
		Speculative Generality	Speculative Generality
	Duplication	Magic Number	n/a
		Duplicated Code	Duplicated Code
		Alternative Classes with Different Interfaces	Alternative Classes with Different Interfaces
	Conditional Logic	Null Check	n/a
		Complicated Boolean Expression	n/a
		Special Case	n/a
		Simulated Inheritance	Switch Statement
Smells Between Classes	Data	Primitive Obsession	Primitive Obsession
		Data Class	Data Class
		Data Clump	Data Clumps
		Temporary Field	Temporary Field
	Inheritance	Refused Bequest	Refused Bequest
		Inappropriate Intimacy (Subclass Form)	Inappropriate Intimacy
		Lazy Class	Lazy Class
	Responsibility	Feature Envy	Feature Envy
		Inappropriate Intimacy (General Form)	Inappropriate Intimacy
		Message Chains	Message Chains
		Middle Man	Middle Man
	Accommodating Change	Divergent Change	Divergent Change
		Shotgun Surgery	Shotgun Surgery
		Parallel Inheritance Hierarchies	Parallel Inheritance Hierarchies
		Combinatorial Explosion	n/a
	Library Classes	Incomplete Library Class	Incomplete Library Class

APPENDIX C – TRIGGER-ELEMENTS OF CODE REFACTORINGS

Columns:

- **No.** – number of a refactoring according to the numbering introduced in Appendix A
- **Refactoring** – name of a refactoring
- **Trigger-element(s)** – name of a trigger-element of a refactoring, its optional multiplicity (in square brackets) and constraint (in curly brackets)
- **Role of trigger-element(s)** – explanation of the role of a trigger-element in a refactoring
- **Trigg**. – "triggerness" of a refactoring – S-T (structure-triggered), B-T (behaviour-triggered)
- **Modifies structure** – YES if a transformation modifies a part of a model that specifies structure of a system, otherwise NO
- **Modifies behaviour** – YES if a transformation modifies a part of a model that specifies behaviour of a system, otherwise NO

No.	Refactoring	Trigger-element(s)	Role of trigger-element(s)	Trigg.	Modifies structure	Modifies behaviour
24.	Change Unidirectional Association to Bidirectional	**Association**	An unidirectional association that will be changed to bidirectional (a field typed by a class on the non-navigable end of the association)	S-T	YES	YES
25.	Change Bidirectional Association to Unidirectional	**Association**	A bidirectional association that will be changed to unidirectional (a field that will be removed)	S-T	YES	YES
13.	Inline Class	**Class**	A class that will be inlined	S-T	YES	YES
14.	Hide Delegate	**Class**	A delegate class	S-T	YES	YES
15.	Remove Middle Man	**Class**	A server class ("middle man")	S-T	YES	YES
20.	Change Value to Reference	**Class**	A class which type will be changed from value to reference	S-T	YES	YES
21.	Change Reference to Value	**Class**	A class which type will be changed from reference to value	S-T	YES	YES
23.	Duplicate Observed Data	**Class**	A presentation class	S-T	YES	YES
40.	Introduce Null Object	**Class**	A source class	S-T	YES	YES
33.	Replace Subclass with Fields	**Class [2..*]**	Subclasses varying only in methods returning constant data	S-T	YES	YES
68.	Replace Delegation with Inheritance	**Class [2]**	A delegate and a delegating class	S-T	YES	YES

No.	Refactoring	Trigger-element(s)	Role of trigger-element(s)	Trigg.	Modifies structure	Modifies behaviour
11.	Move Field	**Field**	A field that will be moved to another class	S-T	YES	YES
18.	Self Encapsulate Field	**Field**	A field that will be encapsulated	S-T	YES	YES
19.	Replace Data Value with Object	**Field**	A field which type will be changed from a primitive type to a class	S-T	YES	YES
27.	Encapsulate Field	**Field**	A field that will be encapsulated	S-T	YES	YES
30.	Replace Type Code with Class	**Field**	A type code field that will be replaced by a class	S-T	YES	YES
31.	Replace Type Code with Subclasses	**Field**	A type code field that will be replaced by a subclasses	S-T	YES	YES
32.	Replace Type Code with State/Strategy	**Field**	A type code field that will be replaced by the state/strategy design pattern	S-T	YES	YES
51.	Remove Setting Method	**Field**	A field that should be set at creation time and never altered	S-T	YES	YES
61.	Push Down Field	**Field**	A field that will be pushed down to subclasses	S-T	YES	NO
57.	Pull Up Field	**Field [2..*]**	Fields that will be pulled up to a superclass	S-T	YES	NO
22.	Replace Array with Object	**Field** {Array}	An array that will be replaced by a class	S-T	YES	YES
28.	Encapsulate Collection	**Field** {Collection}	A collection that will be encapsulated	S-T	YES	YES
65.	Collapse Hierarchy	**Generalization**	Classes that will be joined	S-T	YES	YES
67.	Replace Inheritance with Delegation	**Generalization**	A class and its superclass	S-T	YES	YES
10.	Move Method	**Method**	A method that will be moved to another class	S-T	YES	YES
42.	Rename Method	**Method**	A method that will be renamed	S-T	YES	YES
43.	Add Parameter	**Method**	A method to which a parameter will be added	S-T	YES	YES
52.	Hide Method	**Method**	A method that is not used by any other class	S-T	YES	NO
55.	Replace Error Code with Exception	**Method**	A method that returns an error code	S-T	YES	YES
60.	Push Down Method	**Method**	A method that will be pushed down to subclasses	S-T	YES	NO
17.	Introduce Local Extension	**Method [1..*]**	Foreign methods that will be moved to a local extension	S-T	YES	YES

No.	Refactoring	Trigger-element(s)	Role of trigger-element(s)	Trigg.	Modifies structure	Modifies behaviour
46.	Parameterize Method	**Method [2..*]**	Methods that do similar things but with different values contained in their bodies	S-T	YES	YES
58.	Pull Up Method	**Method [2..*]**	Methods that will be pulled up to a superclass	S-T	YES	NO
53.	Replace Constructor with Factory Method	**Method** {Constructor}	A current constructor	S-T	YES	YES
12.	Extract Class	**Method/Field [1..*]**	Features that will be moved to a new class	S-T	YES	YES
62.	Extract Subclass	**Method/Field [1..*]**	Features that will be extracted to a subclass	S-T	YES	YES
63.	Extract Superclass	**Method/Field [1..*]**	Features that will be extracted to a superclass	S-T	YES	YES
64.	Extract Interface	**Method/Field [1..*]**	Features that will be extracted to an interface	S-T	YES	NO
7.	Remove Assignments to Parameters	**Parameter**	A parameter that will be replaced by a temp	S-T	NO (YES if call by reference)	YES
44.	Remove Parameter	**Parameter**	A parameter that will be removed	S-T	YES	YES
47.	Replace Parameter with Explicit Methods	**Parameter**	An enumerated parameter	S-T	YES	YES
49.	Replace Parameter with Method	**Parameter**	An obsolete parameter	S-T	YES	YES
48.	Preserve Whole Object	**Parameter [2..*]**	Parameters which values are obtained from one class	S-T	YES	YES
50.	Introduce Parameter Object	**Parameter [2..*]**	Parameters that naturally go together	S-T	YES	YES
54.	Encapsulate Downcast	**Parameter** {Return parameter}	A return parameter whose values will be downcasted	S-T	YES	YES
29.	Replace Record with Data Class	**Record**	A record that will be replaced by a class	S-T	YES	YES
1.	Extract Method	**CodeFragment**	A code fragment that will be extracted into a method	B-T	YES	YES
9.	Substitute Algorithm	**CodeFragment**	An algorithm that will be replaced	B-T	NO	YES
41.	Introduce Assertion	**CodeFragment**	A section of code assuming something about the state of the program	B-T	NO/YES (if introduces an assertion class)	YES
59.	Pull Up Constructor Body	**CodeFragment**	A fragment of a constructor body that will be pulled up to the one of a superclass	B-T	YES	YES

No.	Refactoring	Trigger-element(s)	Role of trigger-element(s)	Trigg.	Modifies structure	Modifies behaviour
35.	Consolidate Conditional Expression	**CodeFragment** {Conditional [1..*]}	A sequence of conditional tests that will be extracted into a method	B-T	YES	YES
34.	Decompose Conditional	**CodeFragment** {Conditional}	A conditional that will be simplified by extracting its parts into methods	B-T	YES	YES
36.	Consolidate Duplicate Conditional Fragments	**CodeFragment** {Conditional}	A conditional with duplicated fragments of code	B-T	NO	YES
16.	Introduce Foreign Method	**CodeFragment** {Create expression}	An invocation of a constructor of a server class in the body of a method in a client class	B-T	YES	YES
56.	Replace Exception with Test	**CodeFragment** {Exception}	An exception that will be replaced with a test	B-T	YES	YES
5.	Introduce Explaining Variable	**CodeFragment** {Expression}	An expression that will be replaced by a temp	B-T	NO	YES
38.	Replace Nested Conditional with Guard Clauses	**CodeFragment** {Nested conditional}	A nested conditional that will be replaced with guard clauses	B-T	NO	YES
39.	Replace Conditional with Polymorphism	**CodeFragment** {Switch statement}	A switch statement that will be replaced by polymorphism	B-T	YES	YES
26.	Replace Magic Number with Symbolic Constant	**LiteralNumber**	A literal number that will be replaced by a constant	B-T	NO	YES
2.	Inline Method	**MethodBody**	The body of a method that will be inlined	B-T	YES	YES
8.	Replace Method with Method Object	**MethodBody**	The body of a method that will be turned into a class	B-T	YES	YES
45.	Separate Query from Modifier	**MethodBody**	The body of a method that is both a query and a modifier	B-T	YES	YES
66.	Form Template Method	**MethodBody [2..*]**	Bodes of similar methods	B-T	YES	YES
3.	Inline Temp	**TemporaryVariable**	A temp that will be inlined	B-T	NO	YES
4.	Replace Temp with Query	**TemporaryVariable**	A temp which will be replaced by a method	B-T	YES	YES
6.	Split Temporary Variable	**TemporaryVariable**	A temp that will be split	B-T	NO	YES
37.	Remove Control Flag	**TemporaryVariable**	A control flag for a series of boolean expressions	B-T	NO	YES

APPENDIX D – DEPENDENCIES BETWEEN CODE REFACTORINGS

Columns:

- **No.** – number of a refactoring according to the numbering introduced in Appendix A
- **Refactoring** – name of a refactoring
- **Inverses** – name of an inverse refactoring
- **Includes** – names of transformations that the mechanics of the refactoring includes or may include
- **Is enabled by** – preconditions of the refactoring and names of transformations by which it may be enabled, if these preconditions are not fulfilled
- **Is usually preceded by** – refactorings that usually precede the one
- **Is usually followed by** – refactorings that usually follow the one

The content of the four last columns is based on descriptions of Fowler's refactorings [Fowler *et al.* 1999].

No.	Refactoring	Inverses	Includes	Is enabled by	Is usually preceded by	Is usually followed by
1.	Extract Method	Inline Method (2)		No local-scope variables are modified by the extracted code: - Split Temporary Variable (6) - Replace Temp with Query (4)		
2.	Inline Method	Extract Method (1)				
3.	Inline Temp	Introduce Explaining Variable (5)		The temp is assigned to only once: - Split Temporary Variable (6)		
4.	Replace Temp with Query		Extract Method (1) Inline Temp (3)	The temp is assigned to only once: - Split Temporary Variable (6) Extracted method is free of side effects: - Separate Query from Modifier (45)		
5.	Introduce Explaining Variable	Inline Temp (3)				
6.	Split Temporary Variable					

No.	Refactoring	Inverses	Includes	Is enabled by	Is usually preceded by	Is usually followed by
7.	Remove Assignments to Parameters					
8.	Replace Method with Method Object					
9.	Substitute Algorithm	Substitute Algorithm (9)				
10.	Move Method	Move Method (10)				
11.	Move Field	Move Field (11)		Field is not public: - Encapsulate Field (27) - Self Encapsulate Field (18)		
12.	Extract Class	Inline Class (13)	Move Field (11) Move Method (10)			
13.	Inline Class	Extract Class (12)	Move Method (10) Move Field (11)		A separate interface makes sense for the source class methods – Extract Interface (64)	
14.	Hide Delegate	Remove Middle Man (15)				
15.	Remove Middle Man	Hide Delegate (14)				
16.	Introduce Foreign Method					
17.	Introduce Local Extension		Move Method (10)			
18.	Self Encapsulate Field					
19.	Replace Data Value with Object					You may now use Change Value to Reference (20) on the new object
20.	Change Value to Reference	Change Reference to Value (21)	Replace Constructor with Factory Method (53)			You may want to use Rename Method (42) on the factory to convey that it returns an existing object
21.	Change Reference to Value	Change Value to Reference (20)			The object is immutable – Remove Setting Method (51)	

No.	Refactoring	Inverses	Includes	Is enabled by	Is usually preceded by	Is usually followed by
22.	Replace Array with Object					
23.	Duplicate Observed Data		Self Encapsulate Field (18)			
24.	Change Unidirectional Association to Bidirectional	Change Bidirectional Association to Unidirectional (25)				
25.	Change Bidirectional Association to Unidirectional	Change Unidirectional Association to Bidirectional (24)	Substitute Algorithm (9)			
26.	Replace Magic Number with Symbolic Constant					
27.	Encapsulate Field					
28.	Encapsulate Collection		Rename Method (42) Extract Method (1) Move Method (10)			
29.	Replace Record with Data Class					
30.	Replace Type Code with Class		Rename Method (42)			
31.	Replace Type Code with Subclasses		Self Encapsulate Field (18)		The type code is not passed into the constructor – Replace Constructor with Factory Method (53)	
32.	Replace Type Code with State/Strategy		Self Encapsulate Field (18)			
33.	Replace Subclass with Fields		Replace Constructor with Factory Method (53) Inline Method (2)			
34.	Decompose Conditional		Extract Method (1)			
35.	Consolidate Conditional Expression					Consider using Extract Method (1) on the condition

No.	Refactoring	Inverses	Includes	Is enabled by	Is usually preceded by	Is usually followed by
36.	Consolidate Duplicate Conditional Fragments					If there is more than a single statement, use Extract Method (1) on that code
37.	Remove Control Flag		Extract Method (1)			
38.	Replace Nested Conditional with Guard Clauses					If all guard clauses yield the same result, use Consolidate Conditional Expression (35)
39.	Replace Conditional with Polymorphism		Extract Method (1) Move Method (10)	There is a necessary inheritance structure: - Replace Type Code with Subclasses (31) - Replace Type Code with State/Strategy (32)		
40.	Introduce Null Object					
41.	Introduce Assertion		Extract Method (1)			
42.	Rename Method	Rename Method (42)				
43.	Add Parameter	Remove Parameter (44)				
44.	Remove Parameter	Add Parameter (43)				
45.	Separate Query from Modifier					
46.	Parameterize Method	Replace Parameter with Explicit Methods (47)	Inline Method (2)			
47.	Replace Parameter with Explicit Methods	Parameterize Method (46)				
48.	Preserve Whole Object					
49.	Replace Parameter with Method		Extract Method (1) Remove Parameter (44)			

No.	Refactoring	Inverses	Includes	Is enabled by	Is usually preceded by	Is usually followed by
50.	Introduce Parameter Object		Add Parameter (43) Move Method (10) Extract Method (1)			
51.	Remove Setting Method					
52.	Hide Method					
53.	Replace Constructor with Factory Method					
54.	Encapsulate Downcast		Encapsulate Collection (28)			
55.	Replace Error Code with Exception					
56.	Replace Exception with Test					
57.	Pull Up Field	Push Down Field (61)				Consider using Self Encapsulate Field (18) on the new field
58.	Pull Up Method	Push Down Method (60)	Pull Up Field (57) Self Encapsulate Field (18)		Methods are identical - Substitute Algorithm (9)	
59.	Pull Up Constructor Body					If there is any common code later, use Extract Method (1) to factor out common code and use Pull Up Method (58) to pull it up
60.	Push Down Method	Pull Up Method (58)				
61.	Push Down Field	Pull Up Field (57)				

No.	Refactoring	Inverses	Includes	Is enabled by	Is usually preceded by	Is usually followed by
62.	Extract Subclass	Collapse Hierarchy (65)	Replace Constructor with Factory Method (53) Rename Method (42) Push Down Method (60) Push Down Filed (61) Self Encapsulate Field (18) Replace Conditional with Polymorphism (39) Move Method (10)			
63.	Extract Superclass		Pull Up Field (57) Pull Up Method (58) Pull Up Constructor Body (59) Rename Method (42) Substitute Algorithm (9) Extract Method (1) Form Template Method (66)			
64.	Extract Interface					
65.	Collapse Hierarchy	Extract Subclass (62)	Pull Up Field (57) Pull Up Method (58) Push Down Method (60) Push Down Field (61)			
66.	Form Template Method		Pull Up Method (58) Rename Method (42)			
67.	Replace Inheritance with Delegation	Replace Delegation with Inheritance (68)				
68.	Replace Delegation with Inheritance	Replace Inheritance with Delegation (67)	Rename Method (42)			

APPENDIX E – RESTRICTIONS IN TAU/DEVELOPER 2.4

An excerpt from [Telelogic 2004]

Model Verifier does not support the following UML constructs and use of UML.

Area	Restriction
Classes	• An instance of an active class that has internal structure cannot be dynamically created. • Creation of a passive class is not allowed to be a standalone action. • Passive classes may not contain operations with state implementations. • Passive classes may not realize any interfaces. • Methods of a passive class cannot call methods of an active one. • Multiplicity constraints are not supported for a passive class instance.
Constructors	• A constructor must have a realizing method. • Static constructors and destructors are not allowed. • At most one constructor state machine can be present in a class. • It is not possible to call inherited constructors from a constructor in a passive class.
Destructors	• Active classes should be able to receive a signal that eventually stops the state machine after required clean-up. • In passive classes, destructors cannot call other operations.
Ports	• Ports are not allowed in passive classes. • Ports cannot be redefined.
Operations	• Return value from value-returning operations must be handled in the left hand side of an assignment expression. • A "delete" operation may not be applied to a part. • Redefinition of a finalized operation is not allowed. • Static operations cannot access non-static attributes.
Signals	• Parameters in signal input must not be omitted. • Output via "all" is not supported.
Timers	• Timers must be defined in active classes in which they are used.
Actions	• "Try" actions are not supported. • "Join", "return" and "stop" actions are not allowed inside loops. • "Start Transition" must have a terminating action. • Input "none" is not supported. • "Deep History Nextstate" action is not supported.

Attributes	• Static attributes are not supported.
	• Inheritance of (virtual) operations from a parent class requires the following:
	  ○ Operation attributes must be re-declared in the (finalized) child class,
	  ○ Attributes must however not use the same name in the parent and child operation.
	• *Integer* and *enum* types are not type compatible. Attributes of such types cannot be mixed in expressions and operators.

Miscellaneous	• Multiple inheritance is not supported. • Dynamic creation of instance of a choice is not allowed. • User-defined templates are not supported. • "State" expression is not supported.

APPENDIX F – UML 2.0 & TAU TRIGGER-ELEMENTS

Columns:

- **Trigger-element(s)** – name of a trigger-element of a code refactoring
- **Constraints on trigger-element(s)** – constraints that refine the trigger-element(s)
- **TAU metaclass(es)** – corresponding metaclass(es) of TAU Object Model
- **UML 2.0 metaclass(es)** - corresponding metaclass(es) of the UML 2.0 metamodel

Trigger-element(s)	Constraints on trigger-element(s)	TAU metaclass(es)	UML 2.0 metaclass(es)
Association		*U2Static::**Association***	*Classes::Kernel::**Association***
Class		*U2Static::**Class***	*Classes::Kernel::**Class*** and its increments, namely *CompositeStructures::StructuredClasses::**Class*** and *CommonBehaviors::Communications::**Class***
CodeFragment		*U2Dynamic::**CompoundAction***	*Activities::(Complete)StructuredActivities::**StructuredActivityNode*** or its subclass *Activities::(Complete)StructuredActivities::**SequenceNode***
	Conditional	*U2Dynamic::**IfAction***	*Activities::(Complete)StructuredActivities::**ConditionalNode***
	Create expression	*U2Dynamic::**CreateExpr***	*Actions::BasicActions::**CallOperationAction***
	Exception	*U2Dynamic::**TryAction***	*Activities::ExtraStructuredActivities::**ExceptionHandler***
	Expression	subclasses of *U2Dynamic::**Expression*** (abstract)	*CommonBehaviors::BasicBehaviors::**OpaqueExpression***
	Nested conditional	*U2Dynamic::**IfAction*** (*then* and *else* clauses of *IfAction* are modelled with the use of *Action*, which is a superclass of *the former*)	*Activities::(Complete)StructuredActivities::**ConditionalNode*** (clauses of *ConditionalNode* are modelled with the use of *ActivityNode*, which is one of the superclasses of the former)
	Switch statement	*U2Dynamic::**DecisionAction***	*Activities::(Complete)StructuredActivities::**ConditionalNode***
Field		*U2Static::**Attribute***	*Classes::Kernel::**Property***
	Collection	*U2Static::**Attribute*** (it inherits from *Typed* and thus allows multiplicity bounds to be specified)	*Classes::Kernel::**Property*** (it inherits from *MultiplicityElement* and thus allows multiplicity bounds to be specified)
	Array	*U2Static::**Attribute*** (each *Attribute* that has multiplicity > 1 gets an implicit String/Array collection type)	*Classes::Kernel::**Property*** (*isOrdered* = true; *isUnique* = false)
Generalization		*U2Static::**Generalization***	*Classes::Kernel::**Generalization***
LiteralNumber		*U2Dynamic::**IntegerValue*** or *U2Dynamic::**RealValue***	*Classes::Kernel::**LiteralInteger*** or *Classes::Kernel::**LiteralUnlimitedNatural***
Method		*U2Static::**Operation***	*Classes::Kernel/Interfaces::**Operation*** and its increments, namely *CommonBehaviors::Communications::**Operation*** and *Templates::**Operations***
	Constructor	*U2Static::**Operation*** (*OperationKind* = OkConstructor) having the same name as *Class* that owns it	*e.g. Classes::Kernel/Interfaces::**Operation*** having the same name as a *Class* that owns it
MethodBody		Concrete subclasses of *U2Static::**Method*** (abstract), namely *U2Dynamic::**OperationBody*** and *U2Dynamic::**SimpleStateMachine***	*each subclass of CommonBehaviors::BasicBehaviors::**Behavior*** (abstract) being a method of *Operation* (subclass of *BehavioralFeature*)
Parameter		*U2Static::**Parameter***	*Classes::Kernel::**Parameter***
	Return parameter	*U2Static::**Parameter*** (direction = return)	*Classes::Kernel::**Parameter*** (*direction* = return)
Record		*U2Static::**DataType***	*Classes::Kernel::**DataType***
TemporaryVariable		*U2Static::**Attribute*** owned by e.g. *DefAction*	*Activities::StructuredActivities::**Variable***

APPENDIX G – TAXONOMY OF ATTRIBUTE AND OPERATION IN TAU