Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings

Nikolaos Tsantalis

Ph.D. dissertation

Supervisor Alexander Chatzigeorgiou
Advisors Maria Satratzemi, George Stephanides, Alexander Chatzigeorgiou

Department of Applied Informatics

University of Macedonia
Thessaloniki

August 2010

Summary

Maintenance has gained the most important role in the life cycle of a software product, since it occupies the largest percentage of software development costs. This can be attributed to the fact that a software product should constantly evolve by providing new features, error corrections, performance improvements, and integration of novel technologies in order to remain competitive and diachronically successful. Despite the major importance of software maintenance, the effort being invested by software companies on preventive maintenance (i.e., improvement of design quality in order to increase maintainability) is very limited (lower than 5% of total maintenance costs). This fact indicates that there is a clear need for methods and tools that can be used by the software industry in order to support preventive maintenance, since the manual and human-driven inspection of source code requires tremendous effort and leads to long-term benefits that do not add immediate value to the software product. To this end, this work aims at developing methods and techniques that provide a concrete solution for major design problems whose remedy improves design quality and facilitates increased maintainability. The developed methods face the problem of improving the design quality of an object-oriented system by means of identifying refactoring opportunities which resolve bad smells existing in source code. This refactoring-oriented approach has the ability to produce refactoring solutions which are feasible and behavior preserving by examining a set of preconditions that should apply, pre-evaluate the impact of the identified refactoring opportunities on certain aspects of design quality and provide a ranking of the refactoring solutions allowing the prioritization of maintenance effort on parts of the program that would benefit the most. Therefore, it can be claimed that this work provides the most adequate support for the refactoring process which constitutes a major part of preventive maintenance.

Acknowledgements

I consider a great fortune and honor having Alexander Chatzigeorgiou as mentor and supervisor. His attitude of life constitutes a guide for the pursuit of quality and morality not only in research but also in life. I cannot find a more suitable phrase to express my feelings than the phrase of Alexander The Great for his teacher Aristotle: “I am indebted to my father for living, but to my teacher for living well.”

I owe a lot of gratitude to my family for the emotional and financial support that they offered to me through the years of my studies. I hope that my work and effort makes them proud.

I am very grateful to a person that played a key role in my life, Marina. She always stood by me, providing encouragement and strength to overcome the obstacles and frustrations in this long research journey.

I would like to thank Marios Fokaefs, the “godfather” of JDeodorant, for setting up the base for JDeodorant and participating in its baby development footsteps.

I would like also to thank Anastasios Alexiadis, Marios Fokaefs (again!) and Eric Bouwers for the time and effort that they dedicated in the evaluation of the developed methods. Their feedback was invaluable and helped significantly to improve the proposed methods.

Finally, I would like to thank all the anonymous reviewers that participated in the review process of the proposed methods. Their comments were essential for the formation of the methods presented in this work.

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

The release of a software product does not signal the end of its life cycle. Instead, a software product should constantly evolve by providing new features, error corrections, performance improvements, and integration of novel technologies in order to remain competitive in the demanding and rapidly changing software market. As a result, software maintenance plays the most important role in the survival and diachronic success of a software product. Software maintenance activities can be divided into four distinct categories:

1. *Perfective*: It includes changes which target at improving the software product, such as the addition of new user requirements or the enhancement of performance and usability.
2. *Corrective*: It includes changes which target at the repair of defects and removal of bugs causing an incorrect or unexpected behavior of software.
3. *Adaptive*: It includes changes which target at the adoption of changing environments, such as new operating systems, novel software platforms or frameworks, new programming language features and updated versions of libraries/components.
4. *Preventive*: It includes changes which target at improving the future maintainability and reliability of the software system.

Several empirical studies [68, 79, 107] have shown that more than 75% of total maintenance costs concern adaptive and perfective maintenance. On the other hand, preventive maintenance corresponds to less than 5% of total maintenance costs. This fact clearly illustrates that software companies prefer to invest on reactive maintenance (i.e., perfective, corrective and adaptive) leading to immediate and short-term benefits, rather than proactive maintenance (i.e., preventive) leading to long-term benefits. This fact may also indicate that there is a lack of methods and tools that can be used by the software industry in order to support preventive maintenance.

According to Polo et al. [90] that observed the evolution of software maintenance costs from the early 1970s till the late 1990s based on the results of relevant empirical studies shown in Table 1.1, there is an increasing trend of the proportion of maintenance cost over total software cost. This increasing trend can be attributed to the increased amount of legacy code that has been adapted in order to be integrated with novel technologies and platforms. This phenomenon is referred to as *legacy crisis* [97] to reflect that if the increasing trend of maintenance cost continues, eventually no resources will be left to develop new systems.

Table 1.1: Evolution of the proportion of maintenance cost over total software cost.

Reference	Date	% maintenance cost
Pressman [92]	1970s	35%-40%
Lientz & Swanson [66]	1976	60%
Pigoski [89]	1980-1984	55%
Pressman [92]	1980s	60%
Schach [96]	1987	67%
Pigoski [89]	1985-1989	75%
Frazer [37]	1990	80%
Pigoski [89]	1990s	90%

A major factor affecting the effort required for maintenance tasks is the design quality of software. The importance of design quality is evident in the literature of software engineering through a large variety of books attempting to catalogue good design practices that should be followed in order to build robust, flexible and extensible software systems. Gamma et al. [39] recorded a set of design patterns that can be used as general reusable solutions to commonly occurring problems in software design. Riel [94] recorded a large set of empirical rules that can be used as heuristics in the design process of object-oriented software systems. Finally, Martin [69] formulated a set of principles that should govern the design of object-oriented systems in order to be resilient to changing requirements and require reasonable maintenance effort.

However, it has been observed that the design quality of a software system deteriorates throughout its evolution due to insufficient initial design not being able to facilitate the implementation of novel requirements and *software aging* [87] caused by changes during maintenance being inconsistent with the original design concept. Design quality deterioration manifests itself in the form of design defects or flaws. In the literature of software engineering there are two major works that attempted to systematically record design flaws and propose solutions for their remedy. In the first work, design flaws are referred to as *antipatterns* [19] to reflect that they are actually poor solutions to recurring implementation and design problems, in contrast with design patterns. In the second work, design flaws are referred to as *bad smells* [36] in the sense that they are code fragments whose structure indicates that a design principle is probably violated and need to be restructured. Although antipatterns and bad smells are very closely related concepts they present some major differences. Antipatterns have been used to describe high-level or global design flaws, while bad smells have been used to describe low-level or local design flaws. In other words, an antipattern may be the outcome of multiple bad smells existing in different parts of the code. Due to the locality of bad smells their remedy is equivalent with the application of a single refactoring [36] which can be defined as a code transformation that eliminates the corresponding bad smell without altering the external behavior of the code. On the other hand, the remedy for an antipattern may require the application of several consecutive refactorings or even more drastic solutions such as the redesign of parts of the software. This almost one-to-one mapping of bad smells with refactorings makes their adoption more appealing to the software maintenance community, since their remedy is more straightforward compared to the solutions required for the antipatterns.

1.1 Goal and approach

The goal of this work is to develop methods and techniques that provide a concrete solution for three bad smells which have been recognized by the software maintenance community as major design problems. These bad smells are the following:

- *Feature Envy* [36] bad smell is a sign of violating the design principle of grouping behavior (i.e., methods) with related data (i.e., attributes) and most of the times appears in the form of "a method that is more interested in a class other than the one it actually is in". The effect of this violation is twofold, since a class suffering from such bad smells has low cohesion (i.e., its methods do not operate on common attributes or with each other in order to implement the concept of the class) and at the same time it is coupled with other classes of the program (i.e., its methods use attributes or methods from other classes of the program in order to implement their functionality). As a result, the existence of Feature Envy smells in a class make the class more change-prone and error-prone due to propagation of changes and errors from the classes that it depends on, and decrease its understandability and testability due to the need for understanding, debugging and testing the classes that it depends on in order to implement new features or fix bugs. A solution to this design problem can be given by moving the misplaced methods to the appropriate classes.
- *State or Type Checking* [54, 26] bad smell constitutes a direct violation of the *Open/Closed* principle which states that "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". It is often employed as an alternative approach to polymorphism in order to simulate *late binding* and *dynamic dispatch* and manifests itself as conditional statements that select an execution path either by comparing the value of a variable representing the current state of an object with a set of named constants, or by retrieving the actual subclass type of a reference through *RunTime Type Identification* (RTTI) mechanisms. The effect of this violation is the introduction of additional complexity due to conditional statements consisting of many cases, and code duplication due to conditional statements scattered in many different places of the program that perform state-checking on the same cases for different purposes [36]. As a result, the maintenance of multiple state-checking code fragments operating on common states requires significantly more effort and may introduce consistency errors. A solution to this design problem can be given by replacing the conditional structures with calls to polymorphic methods.
- *Long Method* [36] bad smell manifests itself by means of methods with large size, high complexity and low cohesion among their statements. Such methods require significantly more time and effort for comprehension, debugging, testing and maintenance. A solution to this problem can be given by extracting cohesive parts of a method which implement a distinct functionality into new separate methods.

The developed methods face the problem of improving the design quality of an object-oriented software system by means of identifying refactoring opportunities which resolve bad smells existing in source code. This approach has three advantages over existing approaches:

- It provides a complete solution to the problem of improving the design quality by taking advantage of the fact that bad smells are directly mapped to specific refactoring solutions.
- It has the ability to produce refactoring solutions which are feasible and behavior preserving by examining a set of preconditions that should apply.
- It has the ability to pre-evaluate the effect of the identified refactoring opportunities on design quality, and thus provide a ranking of the refactoring solutions allowing the prioritization of maintenance effort on parts of the program that would benefit the most.

Furthermore, this refactoring-oriented approach covers all distinct activities of the refactoring process which have been defined by Mens and Tourwé [71] as follows:

1. Identify places where software should be refactored (known as bad smells).
2. Determine which refactoring(s) should be applied to the identified places.
3. Guarantee that the applied refactoring preserves behavior.
4. Apply the refactoring.
5. Assess the effect of the refactoring on quality characteristics of the software.
6. Maintain the consistency between the refactored code and other software artifacts (such as documentation, design documents, tests, etc.).

Due to the radical differences in the nature of each design problem, a separate method has been developed based on the specific structural characteristics of each bad smell. The developed methods can be considered as semi-automatic, in the sense that the designer will eventually decide whether an identified refactoring solution should be applied or not based on conceptual or other quality criteria which cannot be inferred from purely structure-based program analysis. Consequently, the results of each method are evaluated by independent experts in order to determine whether the identified refactoring opportunities are conceptually sound, useful and have positive impact on design quality.

1.2 Contribution

The contribution of this thesis lies in the way that faces the problem of improving the design quality of an object-oriented software system. More specifically, it follows an approach that aims at identifying refactoring opportunities which resolve bad smells existing in source code. In this way, it provides a complete solution for the design problems being faced by covering all the activities of the refactoring process [71] including the selection and application of appropriate refactoring transformations, while existing approaches focus only the detection of design problems. Furthermore, it ensures that the extracted refactoring solutions are feasible and behavior preserving by examining a set of preconditions and rules. Additionally, it pre-evaluates the effect of the extracted refactoring solutions on design quality and provides a ranking mechanism allowing the maintainers to prioritize their effort on parts of the program that would benefit the most. As a result, it can be claimed that this approach provides the most adequate support for the refactoring process which constitutes a major part of preventive maintenance.

The developed methods have been implemented as an Eclipse plug-in, named JDeodorant [49] that makes possible their evaluation on real software systems. Furthermore, it can be employed by the software maintenance community in order to perform empirical studies on various research fields.

1.3 Organization

Chapter 2 presents a thorough literature review on the methods which have been proposed for the detection of design flaws or the improvement of design quality, in general. The research works are categorized based on the approach being followed for the solution of the problem. Furthermore, the advantages and limitations of each approach are being discussed.

Chapter 3 presents a method for the detection of Feature Envy bad smells which can be resolved by applying appropriate *Move Method* [36] refactorings. The approach is based on an algorithm that identifies a set of target classes where a given method can be moved to, ranks the target classes according to the number of members being accessed by the given method from each target class, and finally iterates over the sorted set of target classes and selects the first one for which certain preconditions are satisfied as the class where the given method should be moved to. A novel metric, named *Entity Placement* is used in order to pre-evaluate the effect of the identified refactoring opportunities on design quality and present them to the user in a sorted manner according to their impact. The evolution of certain coupling and cohesion metrics when successively applying the refactoring solutions extracted for two open-source projects has shown that the approach is able to identify refactoring opportunities which improve both coupling and cohesion aspects of design quality. The assessment by an independent designer of the refactoring solutions extracted for a system that he developed has shown that the approach is capable of identifying conceptually sound refactoring opportunities.

Chapter 4 presents a technique for the detection of State Checking bad smells which can be resolved by appropriate *Replace Type Code with State/Strategy* and *Replace Conditional with Polymorphism* [36] refactorings. The approach is based on static source code analysis in order to identify problematic conditional structures performing state-checking and group them according to their relevance. Relevant conditional structures are considered those that operate on common named constants or perform *RunTime Type Identification* on subclass types belonging to the same inheritance hierarchy. The philosophy behind this grouping is that the conditional structures belonging to the same group will eventually utilize the same inheritance hierarchy by means of polymorphism. The size of the resulting groups is used as criterion for sorting the corresponding refactoring opportunities. The higher the number of the refactoring opportunities belonging to a given group, the greater the impact of this group on design quality, since the degree of polymorphism (i.e., the number of polymorphic methods added to a single inheritance hierarchy) introduced to the system will be higher. The precision and recall of the approach has been evaluated by comparing the findings of an independent expert to the results of the technique on various open-source projects. Logistic regression analysis has shown that the decision of the independent expert to accept or reject the refactoring opportunities identified by the technique is affected positively by the number of polymorphic methods that will be added to the same inheritance hierarchy and the average number of statements that will be moved to its subclasses, while it is affected negatively by the number of subclasses that will be created.

Chapter 5 presents a method for the detection of Long Method bad smells which can be resolved by applying appropriate *Extract Method* [36] refactorings. The approach is based on two algorithms which aim at finding cohesive code fragments within the body of a given method that implement a distinct functionality. The first algorithm identifies code fragments containing the complete computation of a given variable (*complete computation slice*), meaning that the resulting slice will contain all the assignment statements modifying the value of this variable. The second algorithm identifies code fragments containing all the statements affecting the state of a given object (*object state slice*) by means of method invocations through references pointing to this object. Moreover, a set of rules has been defined that examine whether the identified code fragments can be extracted into separate methods without altering the original program behavior. The assessment by an independent designer of the refactoring opportunities extracted for an open-source project has shown that the approach is capable of identifying slices having a distinct and independent functionality compared to the rest of the method and thus leading to extracted methods with useful functionality, while it also helps significantly to resolve existing design flaws by decomposing complex methods, removing duplicated code among several methods and extracting code fragments suffering from Feature Envy. The evolution of certain slice-based cohesion metrics when applying the refactoring solutions extracted for an open-source project has shown that the approach is able to identify refactoring opportunities which have a positive impact on the cohesion of the decomposed methods and lead to highly cohesive extracted methods. Finally, the success rate of unit testing after the application of the refactoring solutions extracted for an open-source project has shown that the defined behavior preservation rules successfully exclude refactoring opportunities that could possibly cause a change in program behavior.

Chapter 6 presents some design solutions that were adopted in the implementation of JDeodorant Eclipse plug-in [49] in order to employ the Eclipse Java Development Tools (JDT) API in an efficient manner regarding performance and memory requirements. Furthermore, it demonstrates along with code examples the way that the JDT Core and Refactoring Language Toolkit (LTK) APIs were used in order to implement important features of JDeodorant plug-in.

Chapter 7 discusses the evaluation results of the developed methods and provides some general conclusions drawn from this work. Moreover, it discusses ways that the developed methods can be employed for future research.

Chapter 2

2 Literature Review

The problem of improving the design of an already existing object-oriented software system has been faced in the literature with a large variety of different approaches. This chapter presents an attempt to categorize the related work based on the approaches being followed for the solution of the problem and discuss the advantages and limitations of each approach.

2.1 Search-based approach

The works belonging to this category treat the problem of improving object-oriented design by means of refactoring transformations as a search problem in the space of alternative designs. In general, a search-based approach requires:

a. The modeling of a set of refactoring transformations as means to move through the space of alternative designs.
b. The definition of a quality evaluation function (i.e., fitness function) that will serve to rank the alternative designs.
c. The selection and configuration of a search technique that will be used to find the optimal (or a near-optimal) design for the object-oriented system given as input. The optimal solution is a design from which any move to alternative designs does not improve the value of the fitness function.

Eventually, the outcome of such an approach is a sequence of refactoring transformations that should be consecutively applied in order to reach the optimal design in terms of the employed fitness function. Some representative works of this category are the following:

O'Keeffe and Ó Cinnéide [82, 84] proposed an approach for improving an aspect of object-oriented design which is related with the correct utilization of inheritance. As a result, the refactoring transformations used to move through the search space were inheritance-related, such as Push Down Field/Method (i.e., move of a field or method from a superclass to a subclass), Pull Up Field/Method (i.e., move of a field or method from a subclass to a superclass), Extract/Collapse Hierarchy (i.e., introduction/removal of intermediate levels of inheritance), Replace Inheritance with Delegation (i.e., replacement of an inheritance relationship between two classes with a composition relationship) and the reverse. The quality evaluation functions used to rank the alternative designs were based on metrics from the QMOOD hierarchical design quality model [7]. The search techniques used to find the optimal solution were three different versions of a local search algorithm, namely *first-ascent*, *steepest-ascent* and *multiple-restart Hill Climbing*, respectively, and a meta-heuristic technique, namely *Simulated Annealing*. Their approach has been evaluated by comparing the results of the employed search techniques and evaluation functions on two case studies.

O'Keeffe and Ó Cinnéide [83] extended the aforementioned experimental study by using an additional search technique, namely a genetic algorithm, in order to compare the effectiveness of four different search techniques. Their conclusions for each category of search techniques are the following:

- Simulated annealing has the advantage that it is very robust against local optima in the search space, but it requires a large number of parameters making difficult its configuration for any given input. Moreover, the results vary considerably across different input programs and the search is quite slow.
- The genetic algorithm has the advantage that it is easy to establish a set of parameters that work well in the general case, but suffers from the disadvantages that it presents a significantly high computational cost and varies greatly in effectiveness for different input programs.
- Multiple-ascent hill climbing proved to be the most efficient search technique. It is able to produce high-quality results across any input program, has a relatively easy setup of the required parameters, and has significantly faster execution times compared to the rest techniques.
- Steepest-ascent hill climbing is able to produce high quality solutions, but can be considered as rather slow due to its inability to escape local optima.

Seng et al. [98] proposed an approach for improving another aspect of object-oriented design which is related with the cohesion of class modules. The refactoring transformation that was used in order to move through the space of alternative designs was Move Method. The selected fitness function was based on a combination of coupling, cohesion, complexity and stability metrics. A genetic algorithm was employed as search technique where the genomes represent an ordered list of executed refactorings which are updated through mutation and crossover operations. In order to ensure that the resulting genomes contain feasible and behavior preserving refactorings, they have developed a special model that enables the simulation of refactorings by examining all necessary pre- and postconditions.

Harman and Tratt [45] distinguished search-based refactoring approaches into direct and indirect approaches. In the direct approach the program is directly optimized, since the refactoring steps are applied directly to the program, denoting moves from the current design to a near neighbor design in the search space. This approach is best suited to local search techniques, such as hill climbing and simulated annealing, because semantic preservation can only be ensured by applying valid transformations. In the indirect approach the program is indirectly optimized, via the optimization of a sequence of refactoring transformations that should be applied to the program. In this approach the fitness function is computed indirectly, by applying the transformation sequence to the program and measuring the improvement in the metrics of interest. In the indirect approach, it is possible to apply global search techniques, such as genetic algorithms, because the sequence of transformations can be subjected to arbitrary crossover and mutation operations. Furthermore, they applied the concept of Pareto optimality to the problem of class module cohesion optimization. An advantage of Pareto optimality is that it allows to define multiple fitness functions and thus helps to avoid the construction of a single complex function which requires the normalization and weighted combination of several metrics. Moreover, it produces multiple optimal refactoring sequences, allowing the designer to select an appropriate sequence based on his preferences and conceptual criteria.

In general, the disadvantages of search-based approaches are the following:

- The provided solution is non-deterministic, since their outcome is affected by randomness or initial configuration of parameters. This means that the solutions resulting from different executions for the same input program may differ with each other.
- The provided outcome is a sequence of refactoring transformations that should be accepted or rejected by the designer of the examined system in its entirety. This means that search-based approaches do not offer to the designer the opportunity to exclude refactorings which are not conceptually sound, since any deviation from the original refactoring sequence does not lead to the optimal design.
- They require the determination of several arbitrary input parameters in order to operate. Furthermore, the input parameters are based on the specific characteristics of the program given as input for analysis.

2.2 Metric-based approach

The works belonging to this category are based on metrics in order to detect design flaws, antipatterns, or bad smells. Some representative works of this category are the following:

Tahvildari and Kontogiannis [100] used an object-oriented metrics suite consisting of complexity, coupling and cohesion metrics to detect design flaws at class level. In particular, their approach identifies possible violations of design heuristics by assessing which classes of the system exhibit problematic metric values and then selects an appropriate meta-pattern transformation that will potentially improve the corresponding metric values. The first design heuristic being examined is named *key classes* and refers to classes that manage a large amount of other classes or use them in order to implement their functionality and thus exhibit high values of coupling and complexity. The second design heuristic being examined is named *one class - one concept* and its violation refers to classes implementing more than one concepts and thus present low cohesion, or classes implementing a concept that is distributed among many classes and thus are tightly coupled to other classes.

Trifu and Marinescu [104] proposed the concept of *detection strategies* as a means to detect instances of a structural anomaly. A detection strategy is actually a composition of various metric rules (i.e., metrics that should comply with proper threshold values) combined with AND/OR operators into a single rule that expresses a design heuristic. The threshold values used in the metric rules were defined based on statistical data collected from more than 60 Java and 50 C++ projects. The identified design problems can be eliminated based on corresponding *restructuring strategies* which informally describe (i.e., in textual form) the required actions that should be taken for the elimination procedure.

Salehie et al. [95] proposed a metric-based heuristic framework in order to detect design flaws in object-oriented systems. The framework consists of three main components. A *generic object-oriented design knowledge-base* used to store a set of design heuristics, metrics and flaws along with their relationships. A *hot spot indicator* used to point out the most probable defective entities, namely *hot spots*, using primitive classifiers. A *design flaw detector* used to locate possible design flaws in the predetermined hot spots using composite classifiers. The employed classifiers are actually rules based on metrics and fuzzy terms, such as high and low, which are quantified by custom threshold values.

Trifu and Reupke [105] proposed an approach that is based on the idea of combining correlated indicators in order to diagnose certain design flaws, in analogy with the medical world where a disease is diagnosed based on the presence of a specific constellation of symptoms. They distinguish three kinds of indicators, namely aggregating indicators (single metrics or logical expressions combining metrics), structural indicators (patterns in the structure of the code), and semantic indicators (the names of certain program elements, such as variables). The *diagnosis strategy* being followed consists of two parts. The first part automatically produces a set of design flaw instance candidates based on heuristic rules used for checking the required indicators. The second part verifies the semantic context of the design flaw instance candidates based on a set of predefined questions asked to the designer of the system (i.e., the actual design flaw instances are confirmed by the designer of the system).

Moha et al. [74] introduced DECOR, which is a method that embodies all steps required for the specification and detection of code and design smells. Furthermore, they proposed a detection technique, named DETEX, which constitutes an instantiation of the DECOR method. The DETEX technique is comprised of four major steps. *Domain analysis* performs a thorough analysis of the domain related with smells in order to identify key concepts in their text-based descriptions. In this step, a taxonomy and classification of smells is defined based on the key concepts in order to highlight the similarities and differences among smells. The *specification* of smells is performed using a domain-specific language (DSL) in the form of rules using the previous taxonomy. The rules describe the properties that a class must have to be considered a smell. The DSL allows defining properties for the detection of smells, specifying the structural relationships among these properties and characterizing properties according to their lexicon (i.e., names), structure (e.g., classes using global variables), and internal attributes using metrics. The *detection algorithms* are automatically generated by parsing the rules defined in the specification process. Finally, the detection algorithms are automatically applied on a model representation of the system under examination produced during forward engineering or through reverse engineering of its source code.

In general, the disadvantages of metric-based approaches are the following:

- The definition of the threshold values required for the metrics is subjective, since it is arbitrary or based on statistical data. Furthermore, the results can be affected by changing the threshold values.
- They focus on the detection of design flaws without providing specific solutions in terms of refactoring opportunities that can be applied for their remedy.

2.3 Visualization-based approach

The works belonging to this category rely on visualization techniques in order to provide assistance for the detection of design flaws. Some representative works of this category are the following:

Simon et al. [99] defined a cohesion metric based on Jaccard distance in order to quantify the cohesion between attributes and methods. By visual interpretation of the distances between the attributes and methods of a given program in a three-dimensional perspective, the designer can identify methods and attributes which are misplaced in a class other than the one that they should belong to based on access criteria (i.e., the methods and fields that they access or are being accessed from). Such identified cases constitute opportunities for the application of Move Method, Move Field, Extract Class and Inline Class refactorings in order to improve the cohesion and coupling of the program under examination.

Van Emden and Moonen [106] proposed an approach for the automatic detection and visualization of *instanceof* and *typecast* code smells. The *instanceof* code smell appears as a sequence of conditional statements that test an object for its type, while the *typecast* code smell appears when an object is explicitly converted from one class type into another. To this end, they developed a prototype code smell browser, named jCOSMO, which visualizes the detected code smells in the form of a graph. In this graph, the code smells are represented as additional nodes connected to the code entities that they appear in. In this way it is possible to discriminate which parts of the system have the largest number of code smells and would benefit the most from restructuring.

Dhambri et al. [29] proposed a visualization-based approach and a tool that can be used by software maintainers to detect design anomalies. The proposed visualization framework is able to represent four types of information, namely quantitative, architectural, relational and semantic. Quantitative information is captured using metrics extracted from source code. For example, the classes are displayed as 3D boxes and the metrics are associated with three attributes of the 3D box, namely height, color and twist. The height of a 3D box indicates the size of the corresponding class, while the color indicates the position of the corresponding class in the distribution of a specific metric (e.g., classes having an extremely high metric value appear in red color). Architectural information is used to represent the structure of the program in terms of modules/packages. Relational information is used to represent reverse-engineered relationships, such as association, generalization, aggregation and implementation relationships. Finally, semantic information refers to application domain knowledge that can be determined by browsing the source code corresponding to the visualized element.

Parnin et al. [88] proposed a catalogue of lightweight visualizations for design defects being reported by automated inspection tools in order to assist the reviewers in weeding out false positives. The authors divided the code smells according to the code level that they appear into four different categories, namely statement, class, method and collaboration code smells. Although each category of code smells shares some common visualization elements, each code smell is visualized according to its specific needs and characteristics. The authors argue that the direct visualization of code structure or metrics is not sufficient enough for the detection of design problems. Instead, they argue that visualization is more helpful when it is applied on the design flaws which are reported as candidates by code smell detection techniques.

In general, the disadvantages of visualization-based approaches are the following:

- They require significant human intervention for the interpretation of the characteristics and relationships of the visualized elements. As a result, the actual detection of design flaws is completely based on human expertise.
- They are not scalable, since visual interpretation becomes more difficult as the number of visualized elements and attributes increases.

2.4 Logic-based or pattern-based approach

The works belonging to this category use logic programming languages or pattern matching techniques in order to detect refactoring opportunities. In general, this approach requires to express the program under examination and the bad smells to be detected using the same abstract level notation, and then apply logic rules or pattern matching in order to find matches of the bad smell expression into the program expression. Some representative works of this category are the following:

Kataoka et al. [52] proposed an approach for finding refactoring candidates based on program invariants (i.e., pre- and post-conditions). More specifically, they employ the Daikon tool for dynamically discovering invariants at specific program points such as loop heads and procedure entries and exits. Their approach is based on the fact that when a particular pattern of invariant relationships appears at a program point, a specific refactoring is applicable.

Tourwé and Mens [103] employed a logic meta-programming technique to detect bad smells and defined a framework that uses this information to propose adequate refactorings. More specifically, they used SOUL (Smalltalk Open Unification Language), which is a variant of Prolog and combines a declarative language at meta-level with an object-oriented base language, such as Smalltalk. SOUL gives the ability to express base-level programs (i.e., object-oriented programs) as logic terms, facts and rules at the meta-level. Furthermore, meta-level programs (i.e., Prolog-like programs) can manipulate and reason about the structure of the base-level programs. In order to detect *Obsolete Parameter* and *Inappropriate Interfaces* bad smells, the authors defined logic rules at the meta-level that perform a search within the classes of a program and find classes for which the bad smell rules apply. Based on the gathered results, they present to the developer a list of refactorings that can be used to remedy the detected bad smells.

Kosar et al. [58] proposed a context-free grammar in order to transform object-oriented code into a higher abstract level notation, describe bad smells, apply refactorings and check for code consistency after the application of refactorings at context-free grammar level. The bad smells are actually described as patterns using the same grammar notation and an automated pattern matching mechanism is applied in order to detect instances of bad smells.

In general, the disadvantages of logic-based and pattern-based approaches are the following:

- The definition of bad smells requires the knowledge of logic programming languages, grammars or other notations.
- The logic rules or patterns used to define bad smells may not be able to capture actual instances that deviate from standard representation.

2.5 Probabilistic approach

The works belonging to this category depend on probabilistic models in order to classify whether a class presents a design problem or not. Some representative works of this category are the following:

Khomh et al. [55] argued that existing approaches for the detection of design problems do not handle the inherent uncertainty of the detection process (i.e., they do not provide a degree of certainty for each one of the detected design problems). To this end, they proposed an approach based on Bayesian Belief Networks (BBNs) to specify design smells and detect them in programs. Within the context of design smell detection, a BBN is a directed acyclic graph, where nodes correspond to either an input (e.g., a metric value for a given class) if there are no incoming edges, to a decision step if there are incoming edges (e.g., is a class part of a smell given the values of its parent nodes?), or to an output node if there are no outgoing edges. A directed edge between two nodes indicates a probabilistic dependency between the starting and ending nodes. Eventually, the output of a BBN is a probability that a class is part of a design smell. In this way, it is possible to sort the candidate classes for a given design smell and prioritize the inspection of classes with higher probability.

Ananda Rao and Narendar Reddy [4] proposed a quantitative method based on the Design Change Propagation Probability (DCPP) matrix in order to detect two types of bad smells, namely *shotgun surgery* [36] and *divergent change* [36]. These smells are directly involved with the degree of change propagation among the modules of a program. The DCPP matrix for a design of n code artifacts (e.g., classes) is a matrix of n x n size, where entry (i, j) represents the probability that a design change in artifact i requires change in j so as to preserve the overall operation of the program. The authors suggest that if a row in the DCPP matrix contains high probability values with respect to a particular artifact, a change to this artifact will require changes to the artifacts corresponding to the high probability values, indicating the existence of a shotgun surgery smell instance. On the other hand, if a column in the DCPP matrix contains high probability values with respect to a particular artifact, then it can be inferred that this artifact is likely to undergo frequent changes during evolution, indicating the existence of a divergent change smell instance.

A major disadvantage of probabilistic models, like Bayesian networks, is that the required probabilities result based on a training set, which within the context of design smell detection corresponds to metric values for classes which have been already determined as valid instances. Obviously, the training set affects the classification results of the model on the actual data set. On the other hand, an advantage of Bayesian models over other machine learning and statistical models is that they require significantly smaller training sets in order to be effective.

2.6 Remarks

From this literature review of the related work it becomes evident that most research works attempt to propose global solutions covering a large range of design flaws. However, research and practice have shown that there exists no “silver bullet” for the detection of design problems, since each code smell has its own specific structural diversities and thus should be handled in a distinct manner. Furthermore, most research efforts tend to solely focus on the detection of design problems without providing concrete solutions by means of refactoring transformations which are suitable for their remedy. Finally, the vast majority of research works do not provide a quantified estimate of the impact of each detected design flaw (or the potential effect of the refactoring transformations that remedy each detected design flaw) on the overall design quality of the program, not allowing the prioritization of preventive maintenance effort according to maintainability improvement criteria.

Chapter 3

3 Identification of Move Method Refactoring Opportunities

The placement of attributes/methods within classes in an object-oriented system is usually guided by conceptual criteria and aided by appropriate metrics. Moving state and behavior between classes can help to reduce coupling and increase cohesion, but it is non-trivial to identify where such refactorings should be applied. This chapter presents a method for the identification of Move Method refactoring opportunities which resolve a very common manifestation of Feature Envy bad smell. An algorithm that employs the notion of distance between system entities (attributes/methods) and classes extracts a list of behavior-preserving refactorings based on the examination of a set of preconditions.

3.1 Introduction

According to several principles and laws of object-oriented design [39, 69] designers should always strive for low coupling and high cohesion. A number of empirical studies have investigated the relation of coupling and cohesion metrics with external quality indicators. Basili et al. [8] and Briand et al. [14] have shown that coupling metrics can serve as predictors of fault-prone classes. Briand et al. [15] and Chaumun et al. [20] have shown high positive correlation between the impact of changes (ripple effects, changeability) and coupling metrics. Brito e Abreu and Melo [18] have shown that *Coupling Factor* [17] has very high positive correlation with defect density and rework. Binkley and Schach [11] have shown that modules with low coupling (as measured by the *Coupling Dependency Metric*) require less maintenance effort, have fewer maintenance faults and fewer run-time failures. Chidamber et al. [22] have shown that high levels of coupling and lack of cohesion are associated with lower productivity, greater rework and greater design effort. Consequently, low coupling and high cohesion can be regarded as indicators of good design quality in terms of maintenance.

Coupling or cohesion problems manifest themselves in many different ways, with *Feature Envy* bad smell being the most common symptom. Feature Envy is a sign of violating the principle of grouping behavior with related data and occurs when a method is “more interested in a class other than the one it actually is in” [36]. Feature Envy problems can be solved in three ways [36]:

a. By moving a method to the class that it envies (Move Method refactoring).
b. By extracting a method fragment and then moving it to the class that it envies (Extract + Move Method refactoring).
c. By moving an attribute to the class that envies it (Move Field refactoring).

The correct application of the appropriate refactorings in a given system improves its design quality without altering its external behavior. However, the identification of methods, method fragments or attributes that have to be moved to target classes is not always trivial, since existing metrics may highlight coupling/cohesion problems but do not suggest specific refactoring opportunities.

The proposed method considers only Move Method refactoring as solution to the Feature Envy design problem. Moving attributes (fields) from one class to another has not been considered, since this strategy would lead to contradicting refactoring suggestions with respect to the strategy of moving methods. Moreover, fields have stronger conceptual binding to the classes in which they are initially placed, since they are less likely than methods to change once assigned to a class.

In this work, the notion of distance between an entity (attribute or method) and a class is employed, to support the automated identification of Feature Envy bad smells. To this end, an algorithm has been developed that extracts Move Method refactoring suggestions. For each method of the system, the algorithm forms a set of candidate target classes where the method can possibly be moved by examining the entities that it accesses from the system classes (system classes refer to the application or program under consideration excluding imported libraries or frameworks). Then, it iterates over the candidate target classes according to the number of accessed entities and the distance of the method from each candidate class. Eventually, it selects as final target class the first one that satisfies a certain list of preconditions related with the application of Move Method refactorings. The examination of preconditions guarantees that the extracted refactoring suggestions are applicable and preserve the behavior of the code.

Obviously, in large applications several refactoring suggestions may be extracted hindering the designer to assess the effect of each refactoring opportunity. To this end, a novel metric named *Entity Placement* is proposed to rank the refactoring suggestions according to their effect on the design. This metric is based on two principles: a) the distances of the entities belonging to a class from the class itself should be the smallest possible (high cohesion), and b) the distances of the entities not belonging to a class from that class should be as large as possible (low coupling).

The actual application of the refactoring suggestions on source code in order to calculate the Entity Placement metric value that the resulting systems would have can be very time-consuming, especially when the number of suggestions is large. The proposed method offers the advantage of evaluating the effect of a Move Method refactoring without actually applying it on source code. This is achieved by virtually moving methods and calculating the Entity Placement metric.

It should be emphasized that the proposed method is by no means a fully automatic approach. In other words, after the extraction of the refactoring suggestions the designer is responsible for deciding whether a refactoring should be applied or not based on conceptual or other design quality criteria. For example, cases that require the designer's knowledge on the examined system are User Interface methods that should not be moved to classes holding data due to the Model-View-Controller pattern, test methods that should be not moved to the classes being tested, and methods of a composing class that should not be moved to its contained classes due to composition relationships. The tool implementing the proposed method assists the designer to determine the reason for selecting a specific target class over other possible target classes.

The evaluation of the proposed method consists of four parts. The first part contains a qualitative analysis of the refactoring suggestions extracted for an open-source project, along with some interesting insights obtained from the inspection and application of the suggestions. The second part studies the evolution of coupling and cohesion metrics when successively applying the refactoring suggestions extracted for two open-source projects. In the third part of the evaluation, an independent designer provides feedback concerning the conceptual integrity of the refactoring suggestions extracted for the system that he developed. The last part refers to the efficiency of the method based on the computation time required for the extraction of refactoring suggestions on various open-source projects.

3.2 Related Work

Simon et al. [99] defined a distance-based cohesion metric, which measures the cohesion between attributes and methods. This metric aims at identifying methods that use or are used by more features of another class than the class that they belong to, and attributes that are used by more methods of another class than the class that they belong to. The calculated distances are visualized in a three-dimensional perspective supporting the developer to manually identify refactoring opportunities. However, visual interpretation of distance in large systems can be a difficult and subjective task. The approach does not evaluate the effect of each refactoring on the design of the resulting system inhibiting the selection of those refactorings that will actually improve the design. Moreover, the case studies used for demonstrating their approach are small systems written by the authors with very obvious bad smells. The proposed method is inspired by the work of Simon et al. in the sense that it also employs the Jaccard distance. However, the difference lies in the fact that the proposed method defines the distance between an entity (attribute or method) and a class enabling the direct extraction of refactoring suggestions, while the approach of Simon et al. defines the distance between two entities and thus its output requires the application of clustering techniques or visual interpretation in order to extract Move refactoring suggestions to specific classes. To summarize, the proposed method exhibits the following advantages compared to the approach followed by Simon et al.:

1. It clearly indicates which methods and to which class they should be moved.
2. It suggests refactorings which are applicable and behavior-preserving by examining a list of preconditions.
3. It efficiently ranks multiple Move Method refactoring suggestions based on their positive influence on the design of the system.
4. It has been evaluated on real open-source projects.
5. It has been fully automated and implemented as an Eclipse plug-in, allowing the developer to apply the suggested refactorings on source code.

Tahvildari and Kontogiannis [100] used an object-oriented metrics suite consisting of complexity, coupling and cohesion metrics to detect classes for which quality has deteriorated and re-engineer detected design flaws. In particular, they identify possible violations of design heuristics by assessing which classes of the system exhibit problematic metric values and then select an appropriate meta-pattern that will potentially improve the corresponding metric values. A limitation of their approach is that it indicates the kind of the required transformation but does not specify on which specific methods, attributes or classes this transformation should be applied (this process requires human interpretation). Moreover, in case of multiple potential suggestions the approach does not evaluate their effect in order to rank them.

O'Keeffe and Ó Cinnéide [82] treat object-oriented design as a search problem in the space of alternative designs. For this purpose, they employ search algorithms, such as Hill Climbing and Simulated Annealing, using metrics from the QMOOD hierarchical design quality model [7] as a quality evaluation function that ranks the alternative designs. The refactorings used by the search algorithms to move through the space of alternative designs are only inheritance-related (Push Down Field/Method, Pull Up Field/Method and Extract/Collapse Hierarchy).

Seng et al. [98] used a special model that examines a set of pre- and postconditions in order to simulate the application of Move Method refactorings and a genetic algorithm to propose Move Method refactoring suggestions that improve the class structure of a system based on a fitness function. Their approach also includes an initial classification process which excludes from optimization, methods playing special roles in the system's design, such as getter and setter methods, collection accessors, delegation methods, state methods, factory methods and methods participating in design patterns. The approach followed by Seng et al. has the following disadvantages compared to the proposed method:

1. It produces as output a sequence of refactorings that should be applied in order to reach an optimal system in terms of the employed fitness function. If the designer decides not to apply some of the suggested refactorings then the resulting system might be worse than a system resulting from other sequences that have not been presented as solutions to the designer. Moreover, the application of the refactoring suggestions might lead to new refactoring opportunities (not originally present in the initial system), which are not taken into account in the resulting solution. On the contrary, a stepwise approach in which after the application of each refactoring the system is re-evaluated and a new list of refactorings that improve the current system is extracted (including any new refactoring opportunities that might have arisen), provides the possibility to the designer to assess the conceptual integrity of the suggestions at each step. Consequently, the designer is able to determine a sequence of refactoring applications that are conceptually sound and at the same time optimize certain software metrics.
2. It employs a genetic algorithm that makes random choices on mutation and crossover operations and as a result the outcome of each execution on the same input system may differ. Moreover, the outcome depends on initial parameter settings decided by the user. On the contrary, a deterministic approach which suggests refactorings based on Feature Envy criteria always results to the same solution for a certain system.
3. Its efficiency is limited by the following factors: a) it requires numerous generations in order to converge to a solution, b) the algorithm has to be executed several times (10 times for the case study used in the evaluation) in order to gather the common refactoring suggestions from all executions that will be reported as final results, since each execution might lead to different results, c) the algorithm includes in the optimization process all movable methods regardless of whether they suffer from Feature Envy problems or not.
4. It requires the definition of an arbitrary trapezoidal function for the normalization of certain metrics (such as WMC and NOM), a calibration run for optimizing each metric separately and the specification of weights used in the definition of the employed fitness function. On the contrary, the Entity Placement metric does not rely on any arbitrary definitions.

Concerning the evaluation of refactoring effect on design quality the following approaches appear in the literature.

Kataoka et al. [53] proposed a quantitative evaluation methodology to measure the maintainability enhancement effect of refactoring. They defined three coupling metrics (return value, parameter and shared variable coupling) in order to evaluate the refactoring effect. By comparing the metric values before and after the application of refactorings, they evaluate the degree of maintainability enhancement. The definition of each metric contains a coefficient that accounts for inter-class coupling. The coefficient values are based on the specific characteristics of the system under study. However, the authors do not provide a systematic approach for estimating the coefficient values. Moreover, they did not include cohesion as a metric for evaluating the modification of maintainability caused by refactorings.

Du Bois et al. [31] theoretically analyzed the best and worst case impact of refactorings on coupling and cohesion dimensions. The refactorings they studied are Extract Method, Move Method, Replace Method with Method Object, Replace Data Value with Object and Extract Class. According to the authors, moving a method that does not refer to local attributes or methods, or is called upon by only few local methods will increase cohesion. Additionally, moving a method that calls external methods more frequently than it is called will decrease import coupling. These observations are in agreement with the principles on which the proposed method is based.

The cumulative effect of move refactorings (in the sense that their application eventually leads to a system where behavior and data are grouped together properly) could be theoretically also achieved by clustering techniques. However, the object-oriented clustering techniques found in the literature [73, 67] refer to the partitioning and modularization of systems at package level rather than class level. Clustering techniques at class level could possibly lead to an optimal system in terms of coupling and cohesion. However, such techniques would present a solution that is an aggregate of multiple Move Method refactorings which the designer should accept or reject in its entirety. A stepwise approach on the other hand, might not lead to an optimal solution but offers the advantage of gradual change of a system, allowing the designer to assess the conceptual integrity of the refactoring suggestions at each step.

3.3 Method

An object-oriented system is considered to be well-designed in terms of coupling and cohesion when its entities (attributes/methods) are grouped together according to their relevance. During analysis, relevance is usually evaluated on a conceptual basis. However, during design and implementation, relevance can be practically assessed considering the attributes and methods that a method accesses.

The notation required for the formalization of the proposed method is graphically illustrated in the UML class diagram of Figure 3.1. The model represents the types, properties and relationships which are necessary in order to identify Feature Envy bad smells for a given Java program.

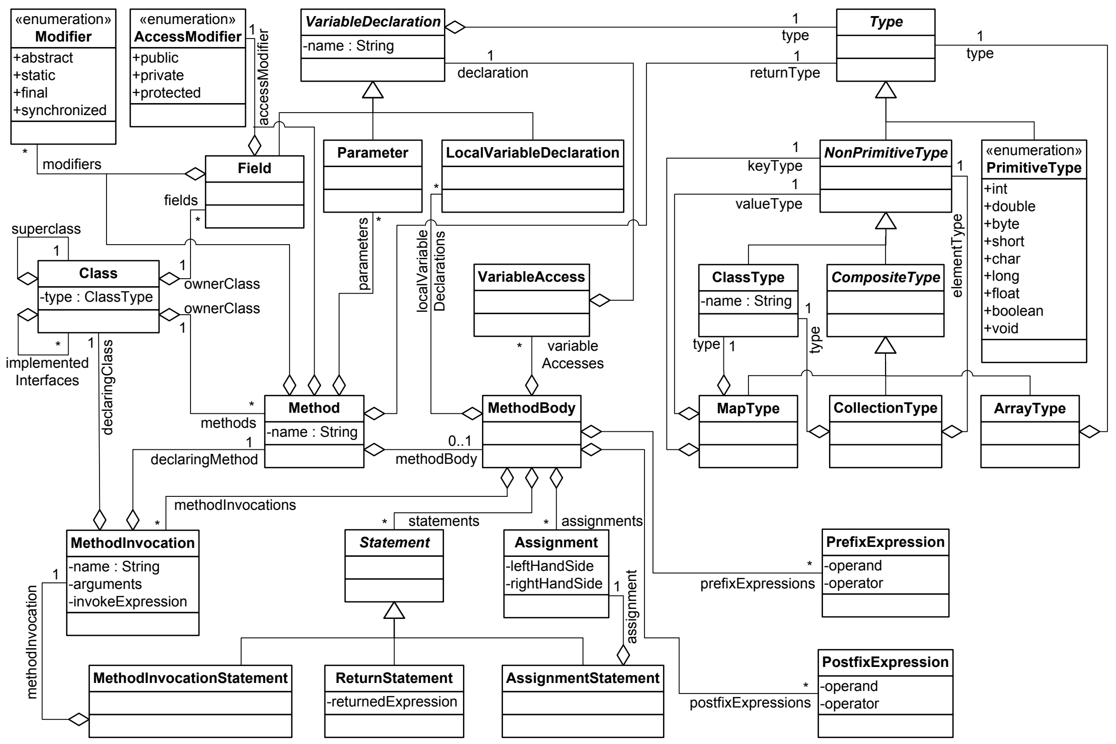


Figure 3.1: UML model for the notation used in the proposed method.

Within the context of the above model, a program has as property the ClassTypes that it contains (denoted as program.classTypes), excluding imported library or framework class types.

3.3.1 Definition of distance

A class in object-oriented programming consists of attributes and methods. Attributes may also be references to other classes of the system (i.e., attributes whose type is a system class), in order to provide access to the functionality of these classes. As a result, a method can access directly attributes and methods of the class that it belongs to and also attributes and methods of other classes through references. Likewise, an attribute can be accessed directly from methods of the class that it belongs to and also from methods of other classes that have a reference to that class.

For each entity (attribute/method), a set of the entities that it accesses (if it is a method) or the entities that it is accessed from (if it is an attribute) is defined.

The entity set of an attribute *attr* contains the following entities:

- the methods directly accessing *attr* that belong to the same class with *attr*
- the methods accessing *attr* that belong to other classes of the system (accesses can be performed either through getter and setter invocations or in exceptional cases directly when *attr* has public visibility)

The entity set of a method *m* contains the following entities:

- the directly accessed attributes that belong to the same class with *m*
- the accessed attributes that belong to other classes of the system
- the directly accessed methods that belong to the same class with *m*
- the accessed methods through reference that belong to other classes of the system

Apart from the entity sets of methods and attributes, the entity set of a class *C* is also defined and contains the following entities:

- all attributes that belong to class *C*
- all methods that belong to class *C*

For the formation of entity sets the following rules should be taken into account. Rules are given in both a descriptive and a formal manner (auxiliary functions are defined in Appendix A):

1. Attributes that are references to classes of the system are not considered as entities nor added to the entity sets of other entities, since such references are essentially a pipeline to the state or behavior of another class.

   (a) if $\exists$ *f* $\in$ *c*.fields where *f*.type $\in$ program.classTypes $\vee$
   (*elementType* = elementTypeOfCollection(*f*) $\neq$ null $\wedge$
   elementType $\in$ program.classTypes)
   then do not add *f* to the entity set of Class *c*

   (b) if $\exists$ *variable* $\in$ *m*.methodBody.variableAccesses
   where *variable*.declaration is Field *f* $\wedge$ (*f*.type $\in$ program.classTypes $\vee$
   (*elementType* = elementTypeOfCollection(*f*) $\neq$ null $\wedge$
   elementType $\in$ program.classTypes))
   then do not add *f* to the entity set of Method *m*

2. Getter and setter methods are not considered as entities nor added to the entity sets of methods and attributes, since they do not offer functionality except for access to attributes. However, the attributes to which they provide access are added to the entity sets. For an attribute that is a collection of objects, the methods that return an element at a specific position, or return an iterator/enumeration of the elements are considered as getters, while the methods that add an element to or replace an element of that collection are considered as setters.

   (a) if $\exists$ *m* $\in$ *c*.methods where (isGetter(*m*) $\neq$ null $\vee$ isSetter(*m*) $\neq$ null $\vee$
   isCollectionGetter(*m*) $\neq$ null $\vee$ isCollectionSetter(*m*) $\neq$ null)
   then do not add *m* to the entity set of Class *c*

   (b) if $\exists$ *methodInv* $\in$ *m*.methodBody.methodInvocations where
   (Field *f* = isGetter(*methodInv*.declaringMethod) $\neq$ null $\vee$
   Field *f* = isSetter(*methodInv*.declaringMethod) $\neq$ null $\vee$
   Field *f* = isCollectionGetter(*methodInv*.declaringMethod) $\neq$ null $\vee$
   Field *f* = isCollectionSetter(*methodInv*.declaringMethod) $\neq$ null)
   then do not add *methodInv*.declaringMethod to the entity set of Method *m*
   if *f* $\neq$ null $\wedge$ (*f*.type $\notin$ program.classTypes $\vee$
   (*elementType* = elementTypeOfCollection(*f*) $\neq$ null
   $\wedge$ *elementType* $\notin$ program.classTypes))
   then add *f* to the entity set of Method *m*

3. Static attributes and methods are not considered as entities nor added to the entity sets of methods and attributes, since they can be accessed or invoked from any method without having a reference to the class that they belong to. An instance method requires the existence of a reference to a target class in order to be moved to that class, and as a result it cannot be moved to a class from which it accesses only static members.

(a) if $\exists f \in c$.fields where f.modifiers $\ni$ `static`
then do not add f to the entity set of Class c

(b) if $\exists\ m \in c$.methods where m.modifiers $\ni$ `static`
then do not add m to the entity set of Class c

(c) if $\exists$ *variable* $\in m$.methodBody.variableAccesses
where *variable*.declaration is Field $f \wedge f$.modifiers $\ni$ `static`
then do not add f to the entity set of Method m

(d) if $\exists$ *methodInv* $\in m$.methodBody.methodInvocations
where *methodInv*.declaringMethod.modifiers $\ni$ `static`
then do not add *methodInv*.declaringMethod to the entity set of Method m

4. Delegate methods are not considered as entities nor added to the entity sets of methods, since they do not offer functionality except for delegating a responsibility to another method. However, the method to which they delegate is added to the entity sets. The treatment of delegations is recursive (in the case of a chain of delegations, only the final non-delegate method is considered).

   (a) if $\exists\ m \in c$.methods where isDelegate(m) $\neq$ null
   then do not add m to the entity set of Class c

   (b) if $\exists$ *methodInv* $\in m$.methodBody.methodInvocations where
   nonDelegateMethod =
   finalNonDelegateMethod(*methodInv*.declaringMethod) $\neq$ null $\wedge$
   (Field f = isGetter(*nonDelegateMethod*) = null $\wedge$
   Field f = isSetter(*nonDelegateMethod*) = null $\wedge$
   Field f = isCollectionGetter(*nonDelegateMethod*) = null $\wedge$
   Field f = isCollectionSetter(*nonDelegateMethod*) = null)
   then add *nonDelegateMethod* to the entity set of Method m
   if $f \neq$ null $\wedge$ (f.type $\notin$ program.classTypes $\vee$
   (*elementType* = elementTypeOfCollection(f) $\neq$ null
   $\wedge$ *elementType* $\notin$ program.classTypes))
   then add f to the entity set of Method m

5. In case of a recursive method, the method itself is not added to its entity set, since a self-invocation does not constitute a dependency with the class that the method belongs to.

   if $\exists$ *methodInv* $\in m$.methodBody.methodInvocations
   where *methodInv*.declaringMethod = m
   then do not add *methodInv*.declaringMethod to the entity set of Method m

6. Access to attributes/methods of classes outside the system boundary (e.g., library classes) is not taken into account. That is because in this approach the library classes are assumed to be fixed from the programmer's perspective and therefore are not subject to refactoring.

The similarity between a method and a class should be high when the number of common entities in their entity sets is large. In order to calculate the similarity of the entity sets the *Jaccard similarity coefficient* is used. For two sets *A* and *B* the Jaccard similarity coefficient is defined as the cardinality of the intersection divided by the cardinality of the union of the two sets:

$$similarity(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The *Jaccard distance* measures the dissimilarity between two sets. For two sets *A* and *B* the Jaccard distance is defined as:

$$distance(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} = 1 - \frac{|A \cap B|}{|A \cup B|} = 1 - similarity(A,B)$$

Let *e* be an entity of the system, *C* a class of the system and S_x the entity set of entity or class *x*. The distance between an entity *e* and a class *C* is calculated as follows:

Definition 1.

if the entity *e* does not belong to the class *C*, the distance is the Jaccard distance of their entity sets:

$$distance(e,C) = 1 - \frac{|S_e \cap S_C|}{|S_e \cup S_C|},\ where\ S_C = \bigcup_{e_i \in C} \{e_i\}$$

Definition 2.

if the entity *e* belongs to the class *C*, *e* is not included in the entity set of class *C*:

$$distance(e,C) = 1 - \frac{|S_e \cap S'_C|}{|S_e \cup S'_C|},\ where\ S'_C = S_C \setminus \{e\}$$

In this way, it is ensured that all distance values range over the interval [0, 1]. If the distance between a class and an entity that belongs to it was calculated without excluding *e* from the entity set of the class, the intersection of their entity sets could never be equal to their union and thus the distance could never obtain the value 0.

3.3.2 Move Method refactoring preconditions

According to Opdyke [86], each refactoring is associated with a set of preconditions which ensure that the behavior of a program will be preserved after the application of the refactoring. In order to describe the preconditions that should be satisfied for a Move Method refactoring in a formal manner, the following set of auxiliary functions is defined:

```
boolean $matchingSignature$(Method $m_1$, Method $m_2$) ≡
    ($m_1$.name = $m_2$.name) ∧
    ($m_1$.returnType = $m_2$.returnType) ∧
    (size of $m_1$.parameters = size of $m_2$.parameters) ∧
    for $i$ = 1 to size of $m_1$.parameters
        $m_1$.parameters[$i$].type = $m_2$.parameters[$i$].type

boolean $abstract$(Method $m$) ≡
    $m$.ownerClass is interface ∨ $m$.modifiers ∋ abstract

(set of Field) $inheritedFields$(Class $c$) ≡
    return $f$ ∈
    {inheritedFields($c$.superclass) ∪ $c$.superclass.fields}
    where $f$.accessModifier ≠ private
```

```
(set of Method) $inheritedMethods$(Class $c$) ≡
    return $m \in$
    {inheritedMethods($c$.superclass) $\cup$ $c$.superclass.methods}
    where $m$.accessModifier $\neq$ private $\wedge$ ~abstract($m$)
```

* In the case where class c does not explicitly inherit a superclass, then its superclass is `java.lang.Object`.

```
(set of Method) $abstractMethodsToBeOverriden$(Class $c$) ≡
    return $m_1 \in$ {abstractMethodsToBeOverriden($c$.superclass) $\cup$
    $c$.superclass.methods} where abstract($m_1$) $\wedge$
    ($\nexists$ $m_2 \in c$.methods $\wedge$ matchingSignature($m_1$, $m_2$))
```

* An abstract method cannot be declared as final, static, or private.

```
(set of Method) $interfaceMethodsToBeImplemented$(Class $c$) ≡
    for $i$ = 1 to size of $c$.implementedInterfaces
        return $m \in$ {interfaceMethodsToBeImplemented($c$.implementedInterfaces[$i$])
        $\cup$ $c$.implementedInterfaces[$i$].methods}
```

* An interface may extend more than one interfaces.

The preconditions that should be satisfied for a Move Method refactoring are divided into three categories, namely compilation preconditions which ensure that the code will compile correctly, behavior-preservation preconditions which ensure that the behavior of the code will be preserved and quality preconditions which ensure that certain design quality properties will not be violated. In all the precondition functions the method parameter (m or m_1) refers to the method to be moved and the class parameter (t) refers to the target class.

3.3.2.1 Compilation preconditions

1. The target class should not contain a method having the same signature with the moved method.

   ```
   $noSimilarLocalMethodInTargetClass$(Method $m_1$, Class $t$) ≡
       $\nexists$ $m_2 \in t$.methods $\wedge$ matchingSignature($m_1$, $m_2$)
   ```

   This issue can be resolved by renaming the moved method.

2. The method to be moved should not override an abstract method. Moving a method that overrides an abstract method would lead to compilation problems, since the overriding of abstract methods is obligatory for concrete classes.

   ```
   $notOverridesAbstractMethod$(Method $m_1$) ≡
       $\nexists$ $m_2 \in$ {abstractMethodsToBeOverriden($m_1$.ownerClass) $\cup$
       interfaceMethodsToBeImplemented($m_1$.ownerClass)} $\wedge$
       matchingSignature($m_1$, $m_2$)
   ```

   This issue can be resolved by keeping the original method as delegate to the moved method.

3. The method to be moved should not contain any super method invocations or super field accesses.

4. The target class should not be an interface, since interfaces contain only abstract methods and not concrete ones.

3.3.2.2 Behavior-preservation preconditions

1. The target class should not inherit a method having the same signature with the moved method. Moving a method which has the same signature with an inherited method of the target class would lead to the overriding of the inherited method, affecting the behavior of the target class and its subclasses.

   noSimilarInheritedMethodInTargetClass(Method m_1, Class t) ≡
   $\nexists$ $m_2 \in$ inheritedMethods(t) $\wedge$ matchingSignature(m_1, m_2)

   This issue can be resolved by renaming the moved method.

2. The method to be moved should not override an inherited method. Moving a method that overrides a concrete method would affect the behavior of the source class and its subclasses, since the source class would inherit the behavior of the method defined in its superclass.

   notOverridesInheritedMethod(Method m_1) ≡
   $\nexists$ $m_2 \in$ inheritedMethods(m_1.ownerClass) $\wedge$ matchingSignature(m_1, m_2)

   This issue can be resolved by keeping the original method as delegate to the moved method.

3. The method to be moved should have a reference to the target class either through its parameters or through source class fields (including inherited fields) of target class type. In order to preserve the behavior of the code, the methods originally invoking the method to be moved should be modified to invoke it through that particular reference after its move. On the contrary, a local variable of target class type declared inside the body of the method to be moved cannot serve as a reference to target class, since it is not accessible outside the method.

   validReferenceToTargetClass(Method m, Class t) ≡
   $\exists$ *variable* $\in$ m.methodBody.variableAccesses where *variable*.declaration $\in$ {m.parameters $\cup$ m.ownerClass.fields $\cup$ inheritedFields(m.ownerClass)} $\wedge$ *variable*.declaration.type = t.type

4. The method to be moved should not be synchronized. The synchronization mechanism of Java ensures that when one thread is executing a synchronized method of an object, all other threads that invoke synchronized methods of the same object suspend the execution until the first thread is done with the object. As a result, the move of a synchronized method could create concurrency problems to the objects of the source class.

3.3.2.3 Quality preconditions

1. The method to be moved should not contain assignments of a source class field (including inherited fields). In that case the assigned field cannot be passed as parameter to the moved method, since parameters are passed by-value in Java and as a result the value of the field will not change after the invocation of the moved method. The alternative approach of passing a parameter of source class type to the moved method and invoking the setter method of the assigned field would increase the coupling between the source and target class, since the moved method

would get coupled to the source class. Moreover, a method that changes the value of a field has stronger conceptual binding with the class where the field belongs to compared to a method that simply accesses the value of the field.

```
noSourceClassFieldAssignment(Method m) $\equiv$
    $\nexists$ assignment $\in$ m.methodBody.assignments
    where assignment.leftHandSide $\in$
    {m.ownerClass.fields $\cup$ inheritedFields(m.ownerClass)} $\wedge$
    $\nexists$ postfixExpression $\in$ m.methodBody.postfixExpressions
    where postfixExpression.operand $\in$
    {m.ownerClass.fields $\cup$ inheritedFields(m.ownerClass)} $\wedge$
    ($\nexists$ prefixExpression $\in$ m.methodBody.prefixExpressions
    where prefixExpression.operand $\in$
    {m.ownerClass.fields $\cup$ inheritedFields(m.ownerClass)} $\wedge$
    (prefixExpression.operator = '+ +' $\vee$ prefixExpression.operator = '– –'))
```

2. The method to be moved should have a one-to-one relationship with the target class. In this way, a method which participates in a one-to-many composition relationship cannot be suggested to be moved from the composing class (the source class that it originally belongs to) to the contained class (target class).

```
one-to-oneRelationshipWithTargetClass(Method m, Class t) $\equiv$
    $\nexists$ variable $\in$ m.methodBody.variableAccesses where variable.declaration $\in$
    {m.ownerClass.fields $\cup$ inheritedFields(m.ownerClass) $\cup$
    m.parameters $\cup$ m.methodBody.localVariableDeclarations} $\wedge$
    ((variable.declaration.type is ArrayType aType $\wedge$ aType.type = t.type) $\vee$
    (elementType = elementTypeOfCollection(variable.declaration) $\neq$ null $\wedge$
    elementType = t.type))
```

3.3.3 Extraction of Move Method refactoring suggestions

The algorithm used for the extraction of Move Method refactoring suggestions is applied to all method entities of a system and consists of four main parts:

1. Identification of the set of candidate target classes T by examining the entity set of method m.
2. Sorting of set T according to the number of entities that method m accesses from each target class in descending order at first level, and according to the distance of method m from each target class in ascending order at second level.
3. Examination of whether method m modifies a data structure in the candidate target classes.
4. Suggestion of moving method m to the first candidate target class that satisfies all the preconditions, following the order of the sorted set T.

It should be noted that the Jaccard distance, which is used for sorting the candidate target classes when the method under examination accesses an equal number of entities from two or more classes, ensures that the candidate target classes will be examined in an order that promotes the classes having fewer entities. This property is desired since it leads to the decomposition of God classes [94] and the equal redistribution of functionality among the system classes. The notion of distance is also employed as a means to rank multiple refactoring suggestions as it will be explained in Section 3.3.4.

The third part of the algorithm aims at identifying cases where the method under examination modifies a data structure in a candidate target class by invoking an appropriate method of the target class and passing as argument one of its parameters. In such a case, it is considered that the method under examination has a strong conceptual binding with the specific target class regardless of the number of entities that the method accesses from the other candidate target classes. For example, in Figure 3.2a method `removeLocation(Location)` has three candidate target classes, namely `TaskManager`, `LocationManager` and `Location`. It accesses one entity from each candidate target class and none from the source class. It invokes method `removeLocation()` through field `locationManager` and passes parameter `loc` as argument to the invoked method. More importantly, method `removeLocation()` of class `LocationManager` (Figure 3.2b) actually removes the passed argument from list `locations` that contains objects of `Location` class type. As a result, class `LocationManager` is considered a better choice for moving the method under examination compared to the other candidate target classes.

```java
public class MyPlannerData {
   private TaskManager tasks;
   private LocationManager locationManager;

   public boolean removeLocation(Location loc) {
      Task[] ts = tasks.tasks();
      for (int i = 0; i < ts.length; i++) {
         if (ts[i].locationID() == loc.id())
            ts[i].setLocationID(
               locations.getLocationAnywhereInstance().id());
      }
      return locationManager.removeLocation(loc);
   }
}
```

(a) Method under examination

```java
public class LocationManager {
   private ArrayList<Location> locations;

   protected boolean removeLocation(Location l) {
      if (!locations.contains(l))
         return false;
      locations.remove(l);
      LocationPair[] lps = getDistancePairs(l);
      for (int i = 0; i < lps.length; i++) {
         distances.remove(lps[i]);
      }
      return true;
   }
}
```

(b) Invoked method belonging to candidate target class

Figure 3.2: Example of method modifying a data structure of a candidate target class.

A formal description of the algorithm used for the extraction of Move Method refactoring suggestions is shown in Figure 3.3.

```
extractMoveMethodRefactoringSuggestions(Method $m$)
    $T = \{\}$
    $S$ = entity set of $m$
    for $i$ = 1 to size of $S$
        $entity = S[i]$
        $T = T \cup \{entity.\text{ownerClass}\}$
    sort($T$)
    $suggestions = \{\}$
    for $i$ = 1 to size of $T$
        if ($T[i] \neq m.\text{ownerClass} \wedge \text{modifiesDataStructureInTargetClass}(m, T[i]) \wedge$
        $\text{preconditionsSatisfied}(m, T[i])$)
            $suggestions = suggestions \cup \{\text{moveMethodSuggestion}(m \rightarrow T[i])\}$
    if $suggestions \neq \varnothing$
        return $suggestions$
    else
        for $i$ = 1 to size of $T$
            if $T[i] = m.\text{ownerClass}$
                return $\{\}$
            else if $\text{preconditionsSatisfied}(m, T[i])$
                return $\{\text{moveMethodSuggestion}(m \rightarrow T[i])\}$
    return $\{\}$
```

Figure 3.3: Algorithm used for the extraction of Move Method refactoring suggestions.

Function *modifiesDataStructureInTargetClass*(Method *m*, Class *t*) which determines whether method *m* modifies a data structure in the candidate target class *t* is formally described in Appendix B. Function *preconditionsSatisfied*(Method *m*, Class *t*) returns true if all preconditions of Section 3.3.2 are satisfied. In the case where method *m* accesses the same number of entities and has the same distance from two or more candidate target classes then suggestions are extracted for all the classes for which the preconditions are satisfied.

3.3.4 Assessing the effect of the refactoring suggestions on design quality

In a large software system it is reasonable to expect that several Move Method refactoring suggestions will be extracted. In that case, it should be possible to distinguish the most effective refactorings in terms of their impact on the design.

The proposed method follows the widely accepted principle of low coupling and high cohesion [40]. To this end, the distances of the entities belonging to a class (inner entities) from the class itself should be the smallest possible (high cohesion). At the same time the distances of the entities not belonging to a class (outer entities) from that class should be as large as possible (low coupling). This can be ensured by considering for each class the ratio of average inner to average outer entity distances. For each class, the closer this ratio to zero is, the safer it can be concluded that inner entities have correctly been placed inside the class and outer entities to other classes. A formula that provides the above information for a class *C* is given by:

$$EntityPlacement_C = \frac{\dfrac{\sum_{e_i \in C} distance(e_i, C)}{|entities \in C|}}{\dfrac{\sum_{e_j \notin C} distance(e_j, C)}{|entities \notin C|}}$$

where *e* denotes an entity of the system. In the special case where a class does not have inner entities the above formula cannot be calculated.

The weighted metric for the entire system which considers the number of entities in each class is given by:

$$EntityPlacement_{System} = \sum_{C_i} \frac{|entities \in C_i|}{|all\ entities|} EntityPlacement_{C_i}$$

The lower the value of this metric is, the more effective is the specific refactoring for the entire system. The classes that do not have inner entities are not included in the above metric.

3.3.5 Virtual application of Move Method refactoring suggestions

In order to evaluate which of the Move Method refactoring suggestions are the most effective ones, one could apply each one of them on source code and then recalculate the distances between the entities and the classes to measure the Entity Placement metric for each of the resulting systems. However, the actual application of the suggested refactorings on source code adds a significant overhead due to disk write operations (once for applying each refactoring and once for undoing it).

To overcome this problem all suggested refactorings are virtually applied. This is achieved by updating the entity sets of the entities/classes which are involved in the move of the corresponding method and calculating the Entity Placement metric for the resulting entity sets.

The virtual move of a method from the source class to a target class is performed as follows:

1. The tag indicating to which class the method belongs is changed from source class to target class.
2. The entity sets of all methods accessing the method are updated according to the new tag.
3. The entity sets of all attributes that are being accessed by the method are updated according to the new tag.
4. The method is removed from the entity set of the source class.
5. The method is added to the entity set of the target class.

The distances which have to be recalculated after the virtual application of a refactoring are: a) the distances from the source and the target class of the entities whose entity set has been affected from the virtual application (i.e., methods that access the moved method and fields being accessed from the moved method), b) the distances from the source and the target class of the entities whose entity set contains at least one entity of the source and/or the target class. The rest of the distances remain unchanged, since the entity sets of the classes that do not participate in the refactoring are the same compared to the initial system.

The extracted refactoring suggestions are ranked in an ascending order according to the corresponding Entity Placement metric values. Eventually, all refactoring suggestions for which the resulting system has a lower Entity Placement value than the current system are considered as refactorings that can improve the design of the system.

3.3.6 Demonstration of the method on a refactoring teaching example

To demonstrate the application of the proposed method, a widely known example for refactorings has been used, namely Fowler's Video Store [36]. The initial version of the program is intentionally not well designed. Its design is gradually improved by applying successive refactorings. A snapshot of the evolving system exactly before the application of the first Move Method refactoring has been taken.

The UML class diagram of the examined snapshot is shown in Figure 3.4. The arrow indicates the move of method `getCharge(Rental)` from class `Customer` to class `Rental`, as suggested by the author of the example.

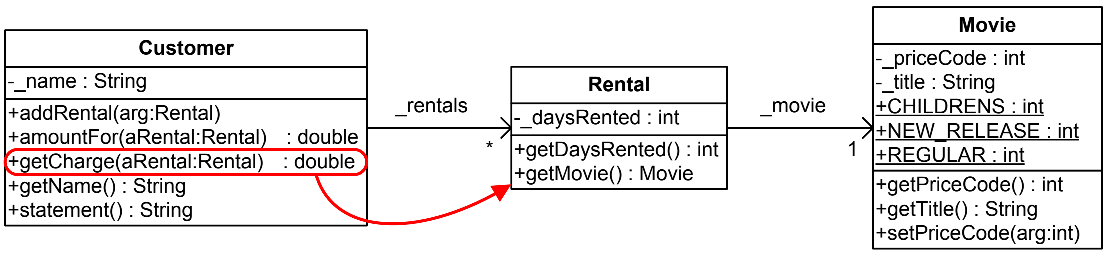


Figure 3.4: UML class diagram of the Video Store before the application of the first Move Method refactoring.

To calculate the distances between the entities and the classes of the system, it is necessary to construct their entity sets as shown in Table 3.1. The entity set of each entity contains the attributes and methods that it accesses (if it is a method) and the methods accessing it (if it is an attribute).

Table 3.1: Information required for extracting the entity sets of all system entities.

Entity name	Accessed attributes	Accessed methods	Accessing methods
Movie::_title	N/A	N/A	Customer::statement()

Movie::_priceCode	N/A	N/A	Customer::statement()
			Customer::getCharge(Rental)

Rental::_daysRented	N/A	N/A	Customer::statement()
			Customer::getCharge(Rental)

Customer::_name	N/A	N/A	Customer::statement()
Customer::statement()	Customer::_name Movie::_priceCode Rental::_daysRented Movie::_title	Customer:: getCharge(Rental)	N/A
Customer::getCharge(Rental)	Movie::_priceCode Rental::_daysRented	-	N/A

N/A: Not Applicable

As it can be observed from Table 3.1 the attributes that are references to classes of the system, namely `Customer::_rentals` and `Rental::_movie` are not considered as entities and do not participate in the entity sets of other system entities. The getter and setter methods of the system, namely `Customer::addRental(Rental)`, `Customer::getName()`, `Rental::getDaysRented()`, `Rental::getMovie()`, `Movie::getPriceCode()`, `Movie::getTitle()`, `Movie::setPriceCode()` are also not considered as entities. However, the attributes to which they provide access (`Customer::_name`, `Rental::_daysRented`, `Movie::_priceCode`, `Movie::_title`) are added to the entity sets of the system entities. The static attributes `Movie::CHILDRENS`, `Movie::NEW_RELEASE` and `Movie::REGULAR` are also not considered as entities and do not participate in the entity sets of the system entities accessing them. Finally, the method `Customer::amountFor(Rental)` that delegates to `Customer::getCharge(Rental)` is not considered as entity and its invocation from method `Customer::statement()` is replaced with the method that it delegates to.

To extract refactoring suggestions for method `Customer::getCharge(Rental)` a set of candidate target classes *T* should be identified by examining its entity set.

The entity set of method `getCharge(Rental)` is:

$$S_{getCharge()} = \{Movie::\_priceCode, Rental::\_daysRented\}$$

and consequently the set of candidate target classes for `getCharge(Rental)` is:

$$T_{getCharge()} = \{Movie, Rental\}$$

Since method `getCharge(Rental)` accesses an equal number of entities from both candidate target classes (i.e., entity *_priceCode* from *Movie* and *_daysRented* from *Rental*), the two candidate target classes will be sorted according to their distance from method `getCharge(Rental)`.

The entity sets of the candidate target classes are the following:

$$S_{Rental} = \{Rental::\_daysRented\}$$

$$S_{Movie} = \{Movie::\_title, Movie::\_priceCode\}$$

The distances between method `Customer::getCharge(Rental)` and the candidate target classes are calculated as:

$$distance(getCharge(), Rental) = 1 - \frac{|S_{getCharge()} \cap S_{Rental}|}{|S_{getCharge()} \cup S_{Rental}|} = 1 - \frac{1}{2} = 0.5$$

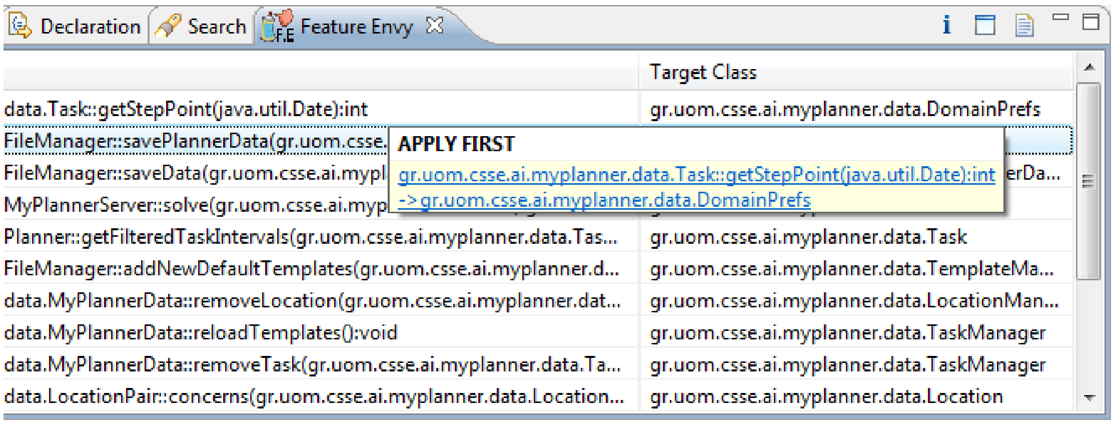
$$distance(getCharge(), Movie) = 1 - \frac{|S_{getCharge()} \cap S_{Movie}|}{|S_{getCharge()} \cup S_{Movie}|} = 1 - \frac{1}{3} = 0.667$$

The target class having the lowest distance from method `getCharge(Rental)` is `Rental` and since all preconditions are satisfied with the specific target class, a Move Method refactoring suggestion is extracted indicating the move of method `get-Charge(Rental)` to class `Rental`. The second candidate target class `Movie` will not be examined by the algorithm since a Move Method refactoring suggestion has been already extracted. However, it should be noted that if class `Movie` was examined as target class the preconditions would not be satisfied since method `get-Charge(Rental)` has a local reference to class `Movie` which is not accessible outside the method.

3.4 JDeodorant Eclipse plug-in

The proposed method has been implemented as an Eclipse plug-in [49] that not only identifies Feature Envy bad smells but also allows the user to apply the refactorings that resolve them on source code. Moreover, the tool pre-evaluates the effect on design quality of all refactoring suggestions, assisting the user to determine the most effective sequence of refactoring applications. The plug-in employs the ASTParser of Eclipse Java Development Tools (JDT) to analyze the source code of Java projects and the ASTRewrite to apply the refactorings and provide undo functionality. JDeodorant offers some novel features concerning the application of Move Method refactorings:

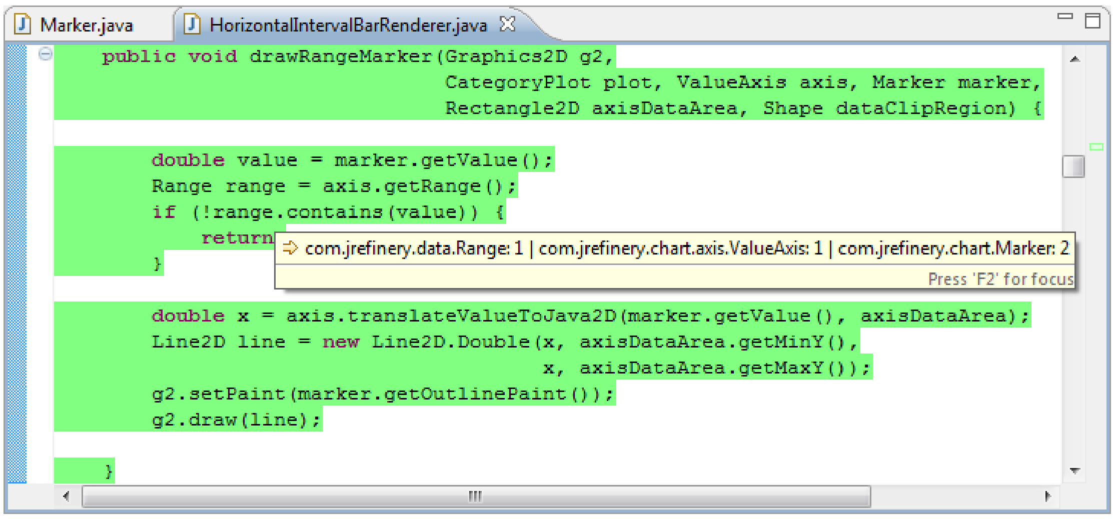
a. It automatically determines whether the original method should be turned into a method that delegates to the moved one. The delegate method is necessary when other classes apart from the source class invoke the method to be moved and prevents these classes from changing the way in which they invoke the moved method.
b. It automatically identifies dependencies between the refactoring suggestions and provides tooltip support aiding the user to resolve them (Figure 3.5). For example, if the method associated with refactoring suggestion *X* invokes a method which is associated with another Move Method refactoring suggestion *Y*, a tooltip informs that suggestion *Y* (corresponding to the invoked method) should be applied before *X* (corresponding to the invoking method).
c. It automatically moves to the target class all the private methods of the source class which are invoked only by the moved method.
d. When the user inspects a method which is suggested to be moved, the tool provides tooltip support indicating the number of members that it accesses from each class (Figure 3.6). In this way the user can more easily realize the Feature Envy problem.



	Target Class
data.Task::getStepPoint(java.util.Date):int	gr.uom.csse.ai.myplanner.data.DomainPrefs
FileManager::savePlannerData(gr.uom.csse.	
FileManager::saveData(gr.uom.csse.ai.mypl	erDa...
MyPlannerServer::solve(gr.uom.csse.ai.myp	
Planner::getFilteredTaskIntervals(gr.uom.csse.ai.myplanner.data.Tas...	gr.uom.csse.ai.myplanner.data.Task
FileManager::addNewDefaultTemplates(gr.uom.csse.ai.myplanner.d...	gr.uom.csse.ai.myplanner.data.TemplateMa...
data.MyPlannerData::removeLocation(gr.uom.csse.ai.myplanner.dat...	gr.uom.csse.ai.myplanner.data.LocationMan...
data.MyPlannerData::reloadTemplates():void	gr.uom.csse.ai.myplanner.data.TaskManager
data.MyPlannerData::removeTask(gr.uom.csse.ai.myplanner.data.Ta...	gr.uom.csse.ai.myplanner.data.TaskManager
data.LocationPair::concerns(gr.uom.csse.ai.myplanner.data.Location...	gr.uom.csse.ai.myplanner.data.Location

APPLY FIRST
gr.uom.csse.ai.myplanner.data.Task::getStepPoint(java.util.Date):int
->gr.uom.csse.ai.myplanner.data.DomainPrefs

Figure 3.5: Tooltip indicating a dependency between two refactoring suggestions.



```
public void drawRangeMarker(Graphics2D g2,
                            CategoryPlot plot, ValueAxis axis, Marker marker,
                            Rectangle2D axisDataArea, Shape dataClipRegion) {

    double value = marker.getValue();
    Range range = axis.getRange();
    if (!range.contains(value)) {
        return;
    }

    double x = axis.translateValueToJava2D(marker.getValue(), axisDataArea);
    Line2D line = new Line2D.Double(x, axisDataArea.getMinY(),
                                    x, axisDataArea.getMaxY());
    g2.setPaint(marker.getOutlinePaint());
    g2.draw(line);

}
```

```
com.jrefinery.data.Range: 1 | com.jrefinery.chart.axis.ValueAxis: 1 | com.jrefinery.chart.Marker: 2
```

Figure 3.6: Tooltip indicating the number of members that the highlighted method accesses from each class.

3.5 Evaluation

The proposed method has been evaluated in four ways:

a. A qualitative analysis of the refactoring suggestions extracted by the proposed method is provided by listing, categorizing and discussing the results for an open-source project.
b. The effect on two aspects of design quality, namely coupling and cohesion is assessed by measuring their evolution when successively applying the suggested refactorings on two open-source projects.
c. Issues regarding conceptual integrity are assessed by requesting from an independent designer to provide feedback on the refactoring suggestions extracted by the proposed method for the system that he developed.
d. The efficiency of the proposed method is evaluated by measuring the computation time with regard to the size of various open-source projects.

For the purpose of evaluation four open-source Java projects have been used, which are relatively active and mature, namely JFreeChart, JEdit, JMol and Diagram.

3.5.1 Qualitative analysis

In order to investigate the kind of Move Method refactoring suggestions extracted by the proposed method, the suggestions have been divided into three main categories, according to the characteristics of the method to be moved:

1. The method does not access any entity from the source class.
2. The method accesses more entities from the target class than the source class.
3. The method accesses an equal number of entities from the source and target classes.

The suggestions belonging to the first category constitute relatively clear cases of Feature Envy. The second and third category refer to cases where the method to be moved has dependency on fields and/or methods of the class that it belongs to. The philosophy behind the suggestions of the third category is that when a method accesses the same number of fields/methods from the source and target classes it should be placed to the smaller class (in terms of the total number of fields/methods), since smaller classes are more easily maintained [16]. Obviously, for all categories the designer should take into account conceptual parameters in order to decide whether the refactoring should be applied or not.

The application of the proposed method to JFreeChart 0.9.6 resulted in 23 Move Method refactoring suggestions leading to a system with lower Entity Placement metric value than the initial system. Table 3.2 contains the source class, method and target class for each suggestion along with the number of members (fields/methods) that the method accesses from the source and target classes. The suggestions are sorted in ascending order according to the corresponding Entity Placement metric values.

Table 3.2: Move Method refactoring suggestions for JFreeChart (version 0.9.6).

id	Source class	Method	Target class	#accessed source members	#accessed target members
1	chart.renderer. HorizontalIntervalBarRenderer	drawRangeMarker	chart.Marker	0	2
2	chart.plot.ContourPlot	drawDomainMarker	chart.Marker	0	3
3	chart.plot.ContourPlot	drawRangeMarker	chart.Marker	0	3
4	chart.StandardLegend	createDrawableLegendItem	chart.LegendItem	1	2
5	chart.axis.DateAxis	previousStandardDate	chart.axis.DateTickUnit	0	3
6	chart.renderer. HorizontalShapeRenderer	drawRangeMarker	chart.Marker	0	2
7	chart.renderer. MinMaxCategoryRenderer	drawRangeMarker	chart.Marker	0	2
8	chart.renderer. VerticalIntervalBarRenderer	drawRangeMarker	chart.Marker	0	2
9	chart.MeterLegend	createLegendItem	chart.LegendItem	0	1
10	data.TimeSeriesCollection	getX	data.RegularTimePeriod	2	3
11	chart.StandardLegendItemLayout	doHorizontalLayout	chart.LegendItemCollection	1	1
12	chart.StandardLegendItemLayout	doVerticalLayout	chart.LegendItemCollection	1	1
13	chart.JFreeChart	drawTitle	chart.AbstractTitle	0	4
14	data.DynamicTimeSeriesCollection	getX	data.RegularTimePeriod	2	3
15	chart.MeterLegend	updateInformation	chart.plot.MeterPlot	0	3
16	chart.plot.PiePlot	getPaint	chart.renderer.PaintTable	2	2
17	chart.axis.DateAxis	nextStandardDate	chart.axis.DateTickUnit	1	2
18	chart.plot.PiePlot	getOutlineStroke	chart.renderer.StrokeTable	2	2
19	chart.plot.PiePlot	getOutlinePaint	chart.renderer.PaintTable	2	2
20	chart.demo.CompassDemo	adjustData	data.DefaultMeterDataset	0	2
21	chart.demo.ThermometerDemo	setMeterValue	data.DefaultMeterDataset	0	3
22	chart.renderer.AbstractRenderer	getSeriesPaint(int, int)	chart.renderer.PaintTable	2	2
23	chart.demo.CompassDemo	pick1PointerAction-Performed	chart.plot.CompassPlot	1	3

* all class names are preceded by package “com.jrefinery.”

To illustrate the soundness of the extracted suggestions the first suggestion of Table 3.2 is analyzed. Method `drawRangeMarker()`, shown in Figure 3.7, does not access any field or method from class `HorizontalIntervalBarRenderer` that it belongs to. The method has 6 parameters in total from which two are not used at all inside the body of the method (`CategoryPlot plot`, `Shape dataClipRegion`), while two other correspond to Java API class types (`Graphics2D g2`, `Rectangle2D axisDataArea`) and thus their types cannot constitute valid target classes. Consequently, method `drawRangeMarker()` has two candidate target classes, namely `ValueAxis` and `Marker`. It invokes two methods of class `Marker` through parameter `marker` and one method of class `ValueAxis` through parameter `axis` and therefore is suggested to be moved to class `Marker`. Moreover, class `Marker` is sufficiently smaller than class `ValueAxis` and constitutes a Data class [36] since it contains only fields and getter methods. This refactoring suggestion is a typical case of moving behavior close to data.

```
public void drawRangeMarker(Graphics2D g2,
      CategoryPlot plot, ValueAxis axis, Marker marker,
      Rectangle2D axisDataArea, Shape dataClipRegion) {

   double value = marker.getValue();
   Range range = axis.getRange();
   if (!range.contains(value)) {
      return;
   }
   double x = axis.translateValueToJava2D(marker.getValue(),
                                          axisDataArea);
   Line2D line = new Line2D.Double(x, axisDataArea.getMinY(),
                                   x, axisDataArea.getMaxY());
   g2.setPaint(marker.getOutlinePaint());
   g2.draw(line);
}
```

Figure 3.7: Method `drawRangeMarker()` corresponding to the first extracted suggestion for JFreeChart (version 0.9.6).

As it can be observed from Table 3.2, in 13 out of 23 refactoring suggestions, the target class belongs to a different package than that of the source class. The suggestion of such kind of refactorings is desirable, since their application may reduce the degree of package dependencies. Moreover, it is harder to identify such refactoring opportunities by manual inspection of the source code, since they require the examination of classes that belong to different packages. Table 3.3 shows the refactoring suggestions belonging to each category.

Table 3.3: Categorization of refactoring suggestions for JFreeChart (version 0.9.6).

category	suggestion ids	#suggestions
1	{1,2,3,5,6,7,8,9,13,15,20,21}	12/23
2	{4,10,14,17,23}	5/23
3	{11,12,16,18,19,22}	6/23

As it can be observed from Table 3.3 about half of the suggestions (12 out of 23) refer to methods that do not access any field or method from the source class. Moreover, 10 out of 11 suggestions belonging to categories 2 and 3 refer to methods that access only fields (no methods) from the source class. In these cases the accessed fields can be passed as parameters to the moved method, resulting in a method that is no longer coupled to the source class. On the contrary, if a method invokes methods of the source class, a parameter of source class type should be added to the moved method in order to be able to invoke them after its move. In this case the moved method remains coupled to the source class. The Entity Placement metric promotes suggestions where methods access only fields from the source class, since such refactorings lead to less coupled methods.

The analysis of the refactorings suggested by the proposed method offered some additional interesting insights:

a. By successively applying the suggested refactorings in JFreeChart 0.9.6 three cases of already existing duplicated code emerged. Specifically, the application of the suggestions 1 and 6 (Table 3.2) resulted in the move of two identical methods named `drawRangeMarker` to class `Marker` that not only had the same signature but also the same body. Two similar cases of duplicated code have been revealed by suggestions 7, 8 and 10, 14 (Table 3.2), respectively. Both pairs of suggestions had as result the move of identical methods (`drawRangeMarker`, `getX`) to a common target class (`Marker`, `RegularTimePeriod`, respectively). Obviously, it is easier for a designer to detect duplicate methods when they exist in the same class, rather than when they are scattered throughout different system classes.

b. The inspection of the refactoring suggestions in JEdit (version 4.3pre12) revealed that several Move Method suggestions were extracted due to the special handling of delegate methods by the proposed method. In the example of Figure 3.8, method `lineComment()` of class `org.gjt.sp.jedit.textarea.TextArea` invokes methods `getLineText()` and `getLineStartOffset()` that delegate to methods of `JEditBuffer` through field `buffer`. Moreover, it accesses 3 methods `rangeLineComment()`, `getSelectedLines()`, `selectNone()` and one field `caret` of class `TextArea`, while it invokes 5 methods of class `JEditBuffer` through field `buffer`. An approach that does not properly handle delegate methods would erroneously consider that method `lineComment()` accesses 6 entities of class `TextArea` and 5 entities of class `JEditBuffer`, thus prohibiting the suggestion of moving the method to class `JEditBuffer`. On the other hand, an approach that properly handles delegate methods would consider that method `lineComment()` accesses 7 entities of class `JEditBuffer` and 4 entities of class `TextArea`.

c. The refactoring suggestions 20, 21 and 23 (Table 3.2) extracted for JFreeChart (version 0.9.6) refer to cases where the source class is a Graphical User Interface (GUI) class which extends class `JPanel` from Java Swing API (`CompassDemo`, `ThermometerDemo`). The corresponding source class methods (`adjustData`, `setMeterValue` and `pick1PointerActionPerformed`) actually modify attributes (through setter methods) of source class fields which can be considered as references to classes holding data. These methods are invoked by ActionListeners which are implemented in the source class and are used to handle ActionEvents on various GUI components (such as buttons and combo boxes) placed on the user interface of the source class. The method suggests that the methods could be moved to the corresponding data classes (`DefaultMeterDataset` and `CompassPlot`). Although these suggestions can be considered as valid in terms of the number of accessed members, they are not conceptually sound since the methods which are related to UI functionality should be separate from the data classes that they may access (according to the Model-View-Controller pattern). This kind of suggestions can be avoided by applying the method separately on the various modules that the system under examination may consist of (e.g., domain classes, GUI classes, database classes, etc.). To this end, the developed tool offers to the designer the possibility of applying the method on a specific package of the examined project. This issue could be also resolved by excluding from examination the methods which are invoked by implemented UI Listener methods (such as method `actionPerformed` of the `ActionListener` interface) using an appropriate precondition.

d. In several of the examined projects, it has been observed that they contain test classes along with the application source code. A test class is responsible for testing whether the behavior of an application class is correct. It usually creates an instance of the class being tested and contains special methods that invoke methods of the tested class with a given input in order to compare the returned result with the expected one. Obviously, the suggestion of moving a test method to the class being tested is not conceptually sound. To this end, the developed tool automatically excludes from the analysis the classes that either extend class `junit.framework.TestCase` from JUnit 3.x API, or contain at least one method annotated with the `@Test` annotation (JUnit 4.x API).

```
protected JEditBuffer buffer;

public final String getLineText(int lineIndex) {
   return buffer.getLineText(lineIndex);
}

public int getLineStartOffset(int line) {
   return buffer.getLineStartOffset(line);
}

public void lineComment() {
   if(!buffer.isEditable()) {
      getToolkit().beep();
      return;
   }
   String comment =
      buffer.getContextSensitiveProperty(caret,"lineComment");
   if(comment == null || comment.length() == 0) {
      rangeLineComment();
      return;
   }
   comment += ' ';
   buffer.beginCompoundEdit();
   int[] lines = getSelectedLines();
   try {
      for(int i = 0; i < lines.length; i++) {
         String text = getLineText(lines[i]);
         buffer.insert(getLineStartOffset(lines[i])
         + StandardUtilities.getLeadingWhiteSpace(text), comment);
      }
   }
   finally {
      buffer.endCompoundEdit();
   }
   selectNone();
}
```
Figure 3.8: Handling of delegate methods in a refactoring suggestion for JEdit (version 4.3pre12).

3.5.2 Evaluation with software metrics

In this part of the evaluation, the most effective refactoring suggestions according to the Entity Placement metric value have been successively applied to open-source projects and the evolution of coupling and cohesion has been studied. The underlying assumption is that refactorings leading to systems with reduced coupling and increased cohesion have a positive effect on design quality. The projects which have been selected for the analysis are JEdit 3.0 (425 classes) and JFreeChart 0.9.6 (459 classes).

There is a wide variety of coupling and cohesion metrics found in the literature [12, 13]. The criterion for choosing the appropriate metrics for the evaluation of the proposed method is that the metrics should be sensitive enough to capture small code changes in an object-oriented system, such as the move of a method from one class to another.

The Message Passing Coupling (MPC) metric [64] has been employed for measuring the evolution of coupling. MPC for a class *C* is defined as the number of invocations of methods not implemented in class *C* by the methods of class *C*. Among the import coupling metrics that consider method-method interactions, MPC evaluates coupling employing the total number of method invocations, while the others measure the number of distinct methods invoked (e.g., RFC [21]). Other more coarse-grained metrics, such as CBO [21] and Coupling Factor [17] have not been considered, because they estimate coupling based on the number of coupled classes and therefore their value might not change when a method is moved.

The redefined Connectivity metric by [12], which has been originally proposed by [46], has been employed for measuring the evolution of cohesion. Connectivity for a class *C* is defined as the number of method pairs of class *C* where one method invokes the other or both access a common attribute of class *C*, over the total number of method pairs of class *C*. Its difference with the other cohesion metrics is that it considers two methods m_1, m_2 to be cohesive, not only if they access a common attribute but also if m_1 invokes m_2 or vice versa. In the implementation of Connectivity metric, the constructors and the accessor (getter and setter) methods have not been considered as methods, since the cohesion of a class is artificially increased if constructors are taken into account and decreased if accessor methods are taken into account [12].

While there is some degree of definitional relevance between Entity Placement and the aforementioned metrics, their major difference lies in the fact that the Jaccard distance (on which Entity Placement is based) is essentially a similarity metric while MPC and Connectivity are based on an absolute count.

Concerning cohesion, Connectivity considers two methods either as cohesive or non-cohesive, while a distance in the numerator of Entity Placement quantifies the degree of similarity between a method and the class to which it belongs. For example, a class might have a Connectivity value of 1(absolute cohesion) because all of its methods invoke each other; however, in the case where these methods invoke also methods from other classes, the numerator of Entity Placement will reveal that the similarity of these methods to the class to which they belong is not absolute (i.e., the average distance is not zero).

Concerning coupling, MPC does not capture the “positive” coupling (expressed by messages being sent from a class to itself), while a distance in the denominator of Entity Placement quantifies for a given class also the similarity of foreign entities from the classes to which they belong. For example, the existence of a method that invokes only methods from its class is not being taken into account in the value of MPC, while it is considered (positively) in the denominator of Entity Placement. Moreover, the MPC metric does not have an upper limit representing the worst case while the worst case for the denominator of Entity Placement occurs when for a given class all foreign entities access all entities from this class and none from the class to which they belong (i.e., the average distance is zero).

The evolution of Entity Placement, MPC and Connectivity for projects JEdit and JFreeChart is shown in Figures 3.9 and 3.10, respectively. At each step the refactoring corresponding to the lowest Entity Placement metric value has been applied. The x-axis represents the successive refactorings that have been performed.

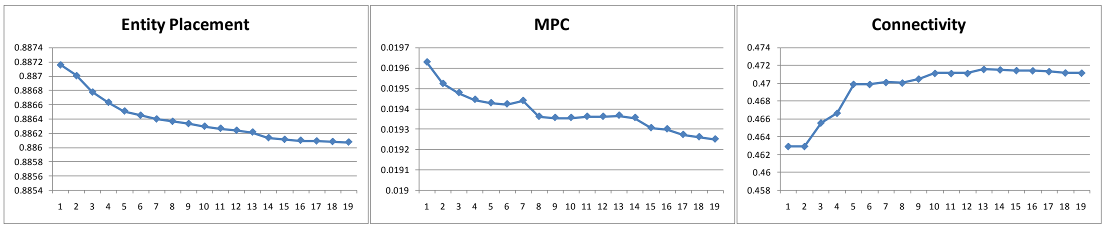


Figure 3.9: Evolution of metrics for JEdit (version 3.0).

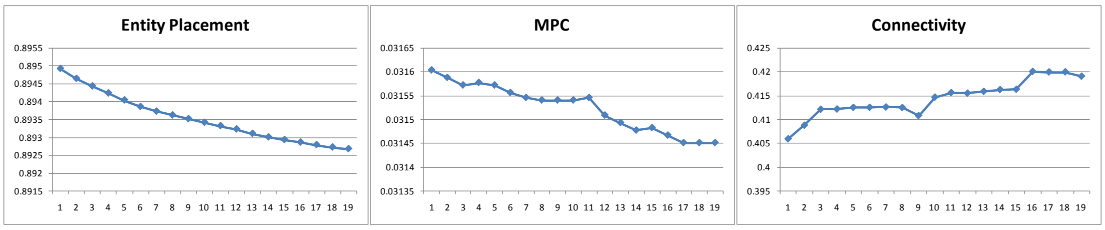


Figure 3.10: Evolution of metrics for JFreeChart (version 0.9.6).

As it can be observed, the application of successive Move Method refactorings, which according to the method reduces the Entity Placement metric value, in general reduces coupling and increases cohesion. The Pearson correlation coefficient between Entity Placement and Message Passing Coupling/Connectivity for the two projects is shown in Table 3.4.

Table 3.4: Correlation between Entity Placement (EP) and Message Passing Coupling (MPC)/Connectivity (Co).

Project	**EP-MPC correlation**	**EP-Co correlation**
JEdit (version 3.0)	0.9572*	-0.9616*
JFreeChart (version 0.9.6)	0.9483*	-0.9246*

* Correlation is significant at the 0.01 level (2-tailed)

The correlation between Entity Placement and coupling, as measured by MPC, is strongly positive and statistically significant for both projects. The correlation between Entity Placement and cohesion, as measured by Connectivity is strongly negative and statistically significant for both projects. Thus, it can be argued that a measure of how well methods and attributes are placed in classes according to the Jaccard distance is a good criterion for ranking Move Method refactoring suggestions.

3.5.3 Independent assessment

In this experiment an independent designer assessed the conceptual integrity of the refactoring suggestions extracted by the proposed method for the system that he developed.

The project that has been examined is called SelfPlanner [93] and is an intelligent web-based calendar application that plans the tasks of a user using an adaptation of the Squeaky Wheel Optimization framework. It is the outcome of a research project of the Artificial Intelligence Group at the department of Applied Informatics, University of Macedonia, Greece. It consists of a planning engine developed in C++ and a client/server application developed in Java. The evaluation focused on the client/server application, since JDeodorant analyzes Java source code. The application (version 1.11) consists of 34 classes and 5800 lines of code. The reasons for selecting the specific project are:

- It is a rather mature research project which has been constantly evolving for more than a year. Moreover, it has been subject to continuous adaptive maintenance due to constant requirement changes. Therefore, it is reasonable to expect that it offers several refactoring opportunities.
- The client/server part of the application was designed and developed by a single person. As a result, the developer that participated in the experiment had complete and deep knowledge of the system's architecture.
- The developer that participated in the experiment is an experienced programmer with knowledge of object-oriented design principles that enabled him to assess the refactoring suggestions extracted by the proposed method and provide valuable feedback.
- The developer that participated in the experiment was able to dedicate a significant amount of time on studying and commenting on the refactoring suggestions extracted by the proposed method.

The refactoring suggestions extracted by the proposed method for SelfPlanner along with the opinion of the independent designer are shown in Table 3.5.

Table 3.5: Move Method refactoring suggestions for SelfPlanner.

id	Source class	Method	Target class	#accessed source members	#accessed target members	designer's opinion
1	data.Task	getStepPoint	data.DomainPrefs	0	2	A
2	FileManager	savePlannerData	Planner	0	4	A
3	FileManager	saveData	data.MyPlannerData	0	1	D
4	MyPlannerServer	solve	Planner	0	3	A
5	Planner	getFilteredTaskIntervals	data.Task	2	4	A
6	FileManager	addNewDefaultTemplates	data.TemplateManager	0	2	A
7	data.MyPlannerData	removeLocation	data.LocationManager	0	1	A
8	data.MyPlannerData	reloadTemplates	data.TaskManager	0	1	A
9	data.MyPlannerData	removeTask	data.TaskManager	0	1	A
10	data.LocationPair	concerns	data.Location	0	1	D

* all class names are preceded by package "gr.uom.csse.ai.myplanner."
A: total agreement, D: conceptual disagreement

As it can be observed from Table 3.5, the independent designer agreed in 8 out of 10 refactoring suggestions. Moreover, 9 out of 10 refactoring suggestions refer to methods that do not access any field or method from the source class and therefore constitute clear cases of Feature Envy. The method corresponding to suggestion 5 accesses two fields from the source class.

Method `saveData()` corresponding to suggestion 3 (Figure 3.11) has one candidate target class, namely `MyPlannerData`. It invokes two methods of the target class through parameter `data` and its purpose is to save the data of the corresponding user as a serialized object. The independent designer supported that the methods which are exclusively related to file operations should be located in class `FileManager` and thus disagreed with this suggestion.

```java
public void saveData(MyPlannerData data) throws IOException {
   data.getTemplateManager().resetTemplates();
   ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream(data.user() + ".spdata", false));
   out.writeObject(data);
   out.close();
}
```
Figure 3.11: Method corresponding to suggestion 3 for SelfPlanner.

Method `concerns()` corresponding to suggestion 10 (Figure 3.12) invokes method `equals()` of class `Location` through two different fields (`l1`, `l2`) that can both serve as reference to target class. By moving this method to class `Location` one of the fields will be replaced with `this` reference and the other will be passed as parameter, resulting in a method with 3 parameters of `Location` type. The independent designer stated that the method will become more complicated if it is moved.

```java
private Location l1;
private Location l2;

public boolean concerns(Location loc1, Location loc2) {
   return ((l1.equals(loc1) && l2.equals(loc2)) ||
           (l1.equals(loc2) && l2.equals(loc1)));
}
```
Figure 3.12: Method corresponding to suggestion 10 for SelfPlanner.

3.5.4 Evaluation of efficiency

The process which is required for the extraction of Move Method refactoring suggestions in a given system consists of the following steps:

a. Parsing of the system under study using the Abstract Syntax Tree (AST) Parser of Eclipse JDT.
b. Determination of system entities and construction of the corresponding entity sets.
c. Calculation of the distances between all system entities and system classes.
d. Application of the algorithm for the extraction of Move Method refactoring suggestions to all method entities of the system. Moreover, all extracted refactoring suggestions are virtually performed in order to calculate the Entity Placement metric value that the system would have if they were actually applied.

Table 3.6 contains various size measures for 4 open-source projects. The measure of examined methods refers to the methods that constitute entities of the system under study. This means that the constructors, accessor methods, static methods and delegate methods are not included in this measure. The measure of total suggestions also includes the refactoring suggestions having a higher Entity Placement value than the initial system. Table 3.7 presents the required computation time for each step of the process.

Table 3.6: Various size measures for the examined open-source projects.

measures	Diagram 1.0.1	JMol 9.0	JEdit 3.0	JFreeChart 0.9.6
#classes	214	316	425	436
#examined methods	727	1435	1864	1847
LOC	23119	49668	71897	106840
#total suggestions	34	246	58	58

Table 3.7: CPU times for each step required for the extraction of refactoring suggestions.

step	Diagram 1.0.1	JMol 9.0	JEdit 3.0	JFreeChart 0.9.6
a	1150 ms	2150 ms	3780 ms	6300 ms
b	30 ms	100 ms	125 ms	80 ms
c	400 ms	2100 ms	3480 ms	3050 ms
d	4.7 sec	133.2 sec	47.1 sec	52 sec

* Measurements performed on Intel Core 2 Duo E6600 2.4 GHz, 2 GB DDR2 RAM

As it can be observed from Table 3.7, the most time-consuming part of the process is the virtual application of the extracted refactoring suggestions. The calculation of the Entity Placement metric value that the system will have after the virtual application of a refactoring suggestion requires a recalculation of distances between the entities and the classes which are affected by the move of the corresponding method. This part can be very time-consuming, since it involves the construction of the union and intersection between several entity sets.

The total CPU time required for the last step primarily depends on the number of the extracted refactoring suggestions. This is evident from the CPU time required for JMol, which has the largest number of refactoring suggestions compared to the other examined systems. The CPU time required for the last step is also affected by the size of the system, since in larger systems more distances have to be calculated. The results satisfy the intuition, since performance is affected by the size of the underlying problem.

Chapter 4

4 Identification of Refactoring Opportunities Introducing Polymorphism

Polymorphism is one of the most important features offered by object-oriented programming languages, since it allows to extend/modify the behavior of a class without altering its source code, in accordance to the Open/Closed Principle. However, there is a lack of methods and tools for the identification of places in the code of an existing system that could benefit from the employment of polymorphism. To this end, a technique that extracts refactoring suggestions introducing polymorphism is proposed. The approach ensures the behavior-preservation of the code and the applicability of the refactoring suggestions based on the examination of a set of preconditions.

4.1 Introduction

Polymorphism has been widely recognized as one of the most important features of object-oriented programming languages. Term polymorphism actually refers to subtype polymorphism which according to Day et al. [24] allows code written in terms of some type *T* to actually work for all subtypes of *T*. The main advantage of polymorphism is that it allows client classes to depend on abstractions [39, 69]. An abstraction (abstract class or interface) can be extended by adding new subclasses that conform to its interface (i.e., override its abstract methods). However, the client classes that depend on abstractions do not have to change in order to take advantage of the behavior defined in the new subclasses.

Despite the sedulous teaching of polymorphism in object-oriented programming courses and its detailed presentation and discussion in books appealing to professionals, *state-checking* is often employed as an alternative approach to polymorphism in order to simulate *late binding* and *dynamic dispatch*. State-checking manifests itself as conditional statements that select an execution path either by comparing the value of a variable representing the current state of an object with a set of named constants, or by retrieving the actual subclass type of a reference through *RunTime Type Identification* (RTTI) mechanisms. The aforementioned symptoms usually result from either poor quality of the initial design or software aging [87] caused by requirement changes that were not anticipated in the original design. State-checking introduces additional complexity due to conditional statements consisting of many cases and code duplication due to conditional statements scattered in many different places of the system that perform state-checking on the same cases for different purposes [36]. As a result, the maintenance of multiple state-checking code fragments operating on common states may require significant effort and introduce errors.

Although the employment of polymorphism in object-oriented systems is considered as an important design quality indicator, there is a lack of tools that either identify state-checking cases in an existing system or eliminate them by applying the appropriate refactorings on source code. To this end, a technique is proposed for the identification and elimination of state-checking problems in Java projects that has been implemented as an Eclipse plug-in. An advantage of the proposed approach over metric-based approaches is the fact that all identified problems are actual cases of state-checking rather than ordinary conditional statements. Moreover, the examination of a set of preconditions ensures that the refactoring suggestions are both applicable and behavior-preserving.

The approach can be considered as semi-automatic, since after the extraction of the refactoring suggestions the designer is responsible for deciding whether a state-checking case should be eliminated or not based on conceptual and design quality criteria. Regarding the automation of the identification process, the main difference of the proposed technique with state-of-the-art Integrated Development Environments (IDEs) offering refactoring support (e.g., Eclipse 3.5, Netbeans 6.7, IntelliJ IDEA 8.1, Visual Studio 2008 along with Refactor! Pro 2.5) is that IDEs determine which refactorings are applicable based on the selection of a code fragment by the developer, while the proposed technique identifies refactoring opportunities without requiring any human intervention. Moreover, the proposed technique assists the designer to determine the effectiveness of the identified refactoring opportunities by grouping them according to their relevance and sorting them according to various quantitative characteristics.

The evaluation of the proposed technique consists of three parts. The first part presents the precision and recall of the approach by comparing the refactoring opportunities identified by an independent expert to the results of the proposed technique on various open-source projects. The second part of the evaluation investigates the impact of three quantitative factors on the decision of the independent expert to accept or reject the refactoring opportunities identified by the proposed technique. The last part refers to the scalability of the technique based on the computation time required for the extraction of refactoring suggestions on various open-source projects which differ in size characteristics.

4.2 Related Work

According to Gamma et al. [39], polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time. To this end, polymorphism plays a key role to the structure and behavior of most design patterns. In the literature of object-oriented software engineering, several empirical studies have investigated the impact of polymorphism and design patterns on external quality indicators related with software maintenance.

Brito e Abreu and Melo [18] have shown that Polymorphism Factor [17], which is defined as the number of methods that override inherited methods divided by the maximum number of possible distinct polymorphic situations, has a moderate to high negative correlation with defect and failure densities as well as with rework. In other words, the appropriate use of polymorphism in an object-oriented design should decrease the defect density and rework. However, they have also supported that very high values of Polymorphism Factor (above 10%) are expected to reduce these benefits, since the understanding and debugging of a highly polymorphical hierarchy is much harder than the procedural counterpart.

Prechelt et al. [91] conducted a controlled experiment to compare design pattern solutions to simpler alternatives in terms of maintenance. The subjects of the experiment were professional software engineers that were asked to perform a variety of maintenance tasks. The independent variables were the programs and change tasks, the program version (there were two different functional equivalent versions of each program, a pattern-based version and an alternative version with simpler solutions) and the amount of pattern knowledge of the participants. The dependent variables were the time taken for each maintenance task and the correctness (i.e., whether the solutions fulfilled the requirements of the task). In most of the cases the experimental results had shown positive effects from the use of design patterns, since maintenance time was reduced compared to the simpler alternative versions.

Ng et al. [77] performed a controlled experiment on maintaining JHotDraw to study whether the introduction of additional patterns through program refactoring is beneficial regardless of the work experience of the maintainers. For this reason, they used two sets of subjects in their experiment, namely experienced and inexperienced maintainers. They compared two maintenance approaches where in the first approach the subjects performed the maintenance tasks directly on the original program, while in the second approach the subjects performed the maintenance tasks on a refactored version of the original program using additional design patterns to facilitate the required changes. The empirical results have shown that, to complete a maintenance task of perfective nature, the time spent even by the inexperienced maintainers on the refactored version was much shorter than that of the experienced subjects on the original version.

Ng et al. [78] studied whether maintainers utilize deployed design patterns, and when they do, which tasks they more commonly perform. For this reason, they refined an anticipated change facilitated by the deployment of design patterns into three finer-grained maintenance tasks, namely adding new concrete participants, modifying the existing interfaces of a participant, and introducing a new client. They concluded that regardless of the type of tasks performed by maintainers when utilizing deployed design patterns for anticipated changes, the delivered code is significantly less faulty than the code developed without utilizing patterns.

Other empirical studies have shown that maintenance effort does not only depend on the design quality of a given program (as expressed by the employment of design principles or the existence of design patterns), but also on human factors such as the experience, skills and education of the software developers and maintainers.

Arisholm and Sjøberg [5] performed a controlled experiment in order to investigate the effect of delegated versus centralized control style on the maintainability of object-oriented software. To this end, two categories of developers (namely experienced and inexperienced) performed several change tasks on two alternative designs that had a centralized and delegated control style, respectively. The results of the experiment have shown that the most experienced developers required less time to maintain the software with delegated control style than with centralized control style, while novice developers had serious problems in understanding the delegated control style and performed much better with the centralized control style. Consequently, they concluded that maintainability of object-oriented software depends, to a large extent, on the skill of the maintainers.

Du Bois [30] performed a series of controlled experiments to investigate whether the application of two reengineering patterns [26], namely *Refactor to Understand*

and *Split Up God Class*, can improve program comprehension. The experiment involving the decomposition of god classes verified that the particular education of the subject performing the comprehension task affects the way in which a god class is decomposed.

Wendorff [109] reported on a large commercial project where the uncontrolled use of patterns has contributed to severe maintenance problems. The reasons causing the maintenance problems were that some pattern instances were misused by software developers who had not understood the rationale behind their employment, many software developers overestimated the future volatility of requirements and opted for patterns to build flexibility at the cost of an undesirable increase of complexity, the change of requirements over the lifetime of the project led some pattern instances to become obsolete, and finally some pattern instances were embellished with additional features which were not actually needed. Consequently, the inappropriate application of patterns may have a negative effect on flexibility and maintainability of object-oriented software.

Concerning performance, it is widely believed that the replacement of conditional logic by a polymorphic method call deteriorates performance due to the introduction of an additional indirection through the *virtual function table*. Demeyer [27] investigated the performance trade-off that is involved when introducing virtual functions by comparing the execution time of four C++ benchmark programs which contain large conditionals against refactored versions where the conditionals were replaced by virtual function calls. The results of the experiment have shown that the optimized code which was generated by three C++ compilers for the refactored versions performed equally or even better compared to the conditional counterparts.

The catalogue of refactorings by Fowler et al. [36] refers to state-checking as the *Switch Statements* bad smell. They argued that the main problem of this smell is code duplication, since the same switch statement is usually scattered in many different places of the code. In such a case, the adaptive maintenance of the code is rather difficult, since the addition of a new clause requires the identification and modification of all these multiple switch statements. The object-oriented paradigm offers the polymorphism mechanism as an elegant way to solve this problem. Fowler et al. proposed two refactorings that eliminate the state-checking code and introduce a new inheritance hierarchy, namely *Replace Type Code with Subclasses* and *Replace Type Code with State/Strategy*. The difference between the two refactorings is that in the first one the inheritance hierarchy is constructed by creating subclasses of the class that originally contained the state-checking code, while in the second one a new *State/Strategy* inheritance hierarchy is created and the class that originally contained the state-checking code becomes the *Context* class in the *State/Strategy* design pattern. It is important to mention that the *Replace Type Code with Subclasses* refactoring is not applicable when the value of the state changes at runtime, since the class type of an object cannot be changed after its creation. Fowler et al. also proposed the *Replace Conditional with Polymorphism* refactoring that eliminates the state-checking code in the case where the inheritance hierarchy already exists.

Demeyer et al. [26] proposed *Reengineering Patterns* as a way to codify and record knowledge about modifying legacy software. Reengineering patterns emphasize on the process of moving from an existing legacy solution to a new refactored solution. Their difference with refactorings is that they also include a process for the detection of the symptoms and a discussion of the impact of changes that the refactored solution may introduce. Within the context of state-checking elimination Demeyer et al. proposed several reengineering patterns which are closely related to specific refactorings. For example, the reengineering patterns *Transform Self Type Checks* and *Transform Client Type Checks* are related to *Replace Type Code with Subclasses* and *Replace Conditional with Polymorphism* refactorings, respectively. Moreover, the reengineering patterns *Factor Out State* and *Factor Out Strategy* are related to *Replace Type Code with State/Strategy* refactoring. The detection process of the symptoms for the aforementioned reengineering patterns is given in the form of guidelines that should be followed by the maintainers in order to manually determine whether a specific conditional statement performs state-checking. Therefore, these guidelines do not constitute a concrete technique that could be automated by means of tools.

Kerievsky [54] proposed a wider set of refactorings as solutions to the design problem of *Conditional Complexity*. The selection of the appropriate refactoring solution depends on the purpose of the conditional logic behind a complex conditional statement. For example, if conditional logic controls the state transitions of an object, then the *Replace State-Altering Conditionals with State* refactoring should be applied. In the case where conditional logic controls which of several variants of a calculation will be executed, then the *Replace Conditional Logic with Strategy* refactoring should be applied. Kerievsky also introduced two novel refactorings that eliminate conditional structures by introducing polymorphism. The first is *Replace Conditional Dispatcher with Command* that breaks down a conditional structure into a collection of Command [39] objects and replaces conditional logic with code to fetch and execute the Command objects. The second is *Move Accumulation to Visitor* that introduces a Visitor [39] in order to remove a conditional structure that is used to obtain data from instances of classes having different interfaces. Although, the author provides a detailed description on the steps required to apply the proposed refactorings (known as mechanics) along with examples from real-world software, the way to identify cases in the code that could benefit from these refactorings is left up to the designer.

Van Emden and Moonen [106] proposed an approach for the automatic detection and visualization of *instanceof* and *typecast* code smells. The *instanceof* code smell appears as a sequence of conditional statements that test an object for its type, while the *typecast* code smell appears when an object is explicitly converted from one class type into another. An interesting part of their approach is the visualization of the detected code smells in the form of a graph, where the code smells are presented as additional nodes connected to the code entities that they belong to. In this way it is possible to discern which parts of the system have the largest number of code smells and would benefit the most from refactoring.

Trifu and Reupke [105] proposed an approach that is based on the idea of combining correlated indicators in order to diagnose certain design flaws, in analogy with the medical world where a disease is diagnosed based on the presence of a specific constellation of symptoms. They distinguish three kinds of indicators, namely aggregating indicators (single metrics or logical expressions combining metrics), structural indicators (patterns in the structure of the code), and semantic indicators (the names of certain program elements, such as variables). Within the context of state-checking they have specified a design flaw named *explicit state checks*. The indicators used for the diagnosis of the specific design flaw are: a) methods that contain “switch” or “if-else-if” conditional structures, and b) checks should be performed on an attribute/property/parameter that semantically indicates the state of the current object instance (i.e., a variable that contains the string "state" in its name). The evaluation of the *explicit state checks* design flaws that were diagnosed in three open-source projects has shown that the employed indicators exhibit low precision when they are triggered individually or even simultaneously.

O'Keeffe and Ó Cinnéide [84] proposed a search-based approach for improving the design of object-oriented programs without altering their behavior. To this end, they formulated the task of design improvement as a search problem in the space of alternative designs. The quality evaluation functions used to rank the alternative designs were based on metrics from the QMOOD hierarchical design quality model. The refactorings used by the search techniques to move through the space of alternative designs were inheritance-related (Push Down Field/Method, Pull Up Field/Method, Extract/Collapse Hierarchy, Replace Inheritance with Delegation, Replace Delegation with Inheritance and many others). Their approach has been validated by two case studies, in which the results of the employed search techniques (Hill Climbing and Simulated Annealing) and evaluation functions have been compared. This work is not directly associated with the elimination of conditional complexity or the introduction of new inheritance hierarchies and polymorphism as a remedy to conditional complexity. However, some of the refactorings used to move through the space of alternative designs may affect the degree of abstraction and polymorphism in a given system. A disadvantage of search-based approaches is that their results rely heavily on the parameterization of the employed search techniques [83].

Ó Cinnéide [80, 81] proposed a method for the automatic introduction of design patterns in terms of refactoring transformations. Based on the observation that design patterns can be decomposed into sequences of *minipatterns* (a minipattern is a design motif that occurs frequently but is a lower-level construct compared to a conventional design pattern), he proposed a set of six reusable *minitransformations* that can define most of the design pattern transformations if composed properly. A minitransformation consists of a precondition, an algorithmic description of the transformation, and a postcondition that ensure behavior preservation. Ó Cinnéide also proposed the concept of *precursor* as a starting point for a design pattern transformation. A precursor is a design structure that serves as an indicator of the need for applying a specific design pattern in future maintenance stages. However, the description of most precursors is by nature quite vague (including the precursor of State pattern) and thus their identification cannot be automated.

4.3 Technique

Let us consider that a state-checking code fragment exists inside the body of method *m* belonging to class *C*. The following sets will be used for the description of the proposed technique:

IV_C : the set of Instance Variables (non-static fields) of class *C*

M_C : the set of non-static Methods of class *C*

P_m : the set of Parameters of method *m*

LV_m : the set of Local Variables declared inside the body of method *m* and before the state-checking code fragment

NC : the set of Named Constants of all system classes (static final fields of `int`, `short`, `char`, or `byte` type and `enum` constant declarations)

4.3.1 Identification of conditional structures performing state-checking

A code fragment that performs state-checking based on named constants can be either a `switch` statement or an `if/else if` statement (each `if` statement should be the `else` clause of the previous `if` statement, except for the first one).

The set of candidate State Variables *SV* (variables that can possibly hold a value representing the current state) is the subset of variables from the union of IV_C, P_m and LV_m sets having `int`, `short`, `char`, `byte`, or `enum` type:

$SV = \{x \in \{IV_C \cup P_m \cup LV_m\}: type_x = primitive \vee type_x = enum\}$

In the case where the state-checking code fragment under study is a `switch` statement *s* consisting of *n* cases the following conditions should be satisfied:

1. The expression of *s* should be a variable *v* belonging to *SV* or an invocation of the getter method of a variable *v* belonging to *SV* provided that *v* is an instance variable.
2. For each `switch case` *c* of *s*, the expression of *c* should be a named constant belonging to *NC*.
3. *n* should be greater than one, or equal to one provided that a `default case` exists.

In the case where the state-checking code fragment under study is an `if/else if` statement consisting of *n* `if` statements the following conditions should be satisfied:

1. The expression of each `if` statement should be (or should contain a conditional sub-expression that is) an infix expression with equality operator. Moreover, one of the operands should be a variable *v* belonging to *SV* or an invocation of the getter method of a variable *v* belonging to *SV* provided that *v* is an instance variable, while the other operand should be a named constant belonging to *NC*.
2. Variable *v* should be common in all infix expressions of the *n* `if` statements.
3. *n* should be greater than one, or equal to one provided that a final `else` clause exists.

If all conditions are satisfied the following information is extracted:

a. the state variable *v*
b. the set of Identified Named Constants (*INC*) that participate in the specific state-checking code fragment along with the code corresponding to each named constant

It is quite possible that the set of identified named constants *INC* for a specific state-checking code fragment does not contain all the named constants that the state variable *v* is actually related with. The identification of all relevant states (represented by named constants) is very important in order to create a State inheritance hierarchy that can be also utilized by other state-checking code fragments which may operate on different but relevant named constants. Otherwise, it could be possible to generate multiple inheritance hierarchies that constitute different concrete implementations of essentially the same state abstraction, causing serious design flaws.

The proposed technique follows two complementary approaches in order to identify the set of Additional Named Constants (*ANC*) which are conceptually related with the state variable *v* but do not participate in the specific state-checking code fragment. The set of additional named constants *ANC* is also added to the information that is extracted for each state-checking code fragment along with the code corresponding to the `default case` or final `else` clause.

Approach based on the state variable v:

a. If variable *v* is an instance variable of class *C*, then all methods of class *C* are examined for assignments where the left hand side is variable *v*.
b. If variable *v* is a local variable or parameter of method *m*, then only method *m* is examined for assignments where the left hand side is variable *v*.

If the right hand side of these assignments is a named constant belonging to *NC* set but not to *INC* set (*NC* \ *INC*), then the named constant is added to the *ANC* set.

Approach based on the sets of identified named constants INC:

The concept behind this approach is that if two state-checking code fragments operate on at least one common named constant (the intersection of their *INC* sets is not empty), then their state variables are conceptually related with all the named constants belonging to the union of their *INC* sets. Thus, a single State inheritance hierarchy should be created for both state-checking code fragments having as many concrete state subclasses as the named constants belonging to the union of their *INC* sets.

To this end, an algorithm (Figure 4.1) is proposed that identifies the maximum number of conceptually relevant named constants by searching for common named constants among the *INC* sets of all state-checking code fragments. Let us consider that INC_i is the set of identified named constants for state-checking code fragment *i*. The *INC* sets are sorted in a list *INCList* according to their cardinality in descending order. The examination of the *INC* sets in descending order increases the probability of identifying a larger number of conceptually relevant named constants (in a single iteration) compared to a random order. A set of named constants *namedConstants* is used to temporarily store the conceptually relevant named constants, while a set of indexes *indexSet* is used to temporarily store the indexes of *INC* sets which have common elements.

```
while INCList not empty
    namedConstants = $INC_0$
    indexSet = {0}
    do
        previousCardinality = |namedConstants|
        for(i=1; i<size of INCList; i++)
            if(namedConstants $\cap$ $INC_i \neq \emptyset$)
                namedConstants = namedConstants $\cup$ $INC_i$
                indexSet = indexSet $\cup$ {i}
    while(previousCardinality < |namedConstants|)

    for each index j in indexSet
        $ANC_j$ = namedConstants \ $INC_j$
        remove $INC_j$ from INCList
```

Figure 4.1: Algorithm for the identification of relevant named constants.

The proposed approach is unable to identify named constants which conceptually belong to a group of relevant states but do not participate in any conditional structure performing state-checking (i.e., they do not belong to the union of all *INC* sets) or are not assigned to any variable holding the current state (state variable *v*). Usually, such named constants express possible states or types that will become active in a future software release or are already active states whose functionality is covered by the `default case` or final `else` clause of the related state-checking code fragments (and thus they do not participate directly in state-checking). Obviously, such cases of named constants require human intervention to be discovered.

4.3.2 Identification of conditional structures performing RTTI

A code fragment that performs RunTime Type Identification can be an `if/else if` statement (each `if` statement should be the `else` clause of the previous `if` statement, except for the first one) consisting of *n* `if` statements.

The set of candidate Superclass Type Variables (*STV*) is the subset of variables from the union of IV_{C}, P_{m} and LV_{m} sets having the type of a system class which is inherited by other classes of the system.

A valid case of RunTime Type Identification should satisfy the following conditions:

1. The expression of each `if` statement should be (or should contain a conditional sub-expression that is) either:
   a. An `instanceof` expression where the left operand is a variable *v* belonging to *STV*, while the right operand is a class type that inherits the superclass type corresponding to variable *v*.
   b. An infix expression with equality operator where one of the operands is the invocation of method `getClass()` over a variable *v* belonging to *STV* (*v.getClass()*), while the other operand is a type literal (*Type.class*) which is a subclass of the superclass type corresponding to variable *v*.
2. Variable *v* should be common in all expressions of the *n* `if` statements.
3. *n* should be greater than one, or equal to one provided that a final `else` clause exists.

If all conditions are satisfied the following information is extracted:

a. the variable *v* (reference to superclass type)
b. the set of Identified Subclass Types (*IST*) that participate in the specific `if/else if` statement along with the code corresponding to each subclass type
c. the inheritance hierarchy tree structure corresponding to the identified class types
d. the code corresponding to the final `else` clause

4.3.3 Handling of compound conditional expressions

In the case where the expression of an `if` statement consists of sub-expressions combined with conditional AND operators (&&), the proposed technique identifies which sub-expression actually performs state-checking and constructs a new conditional expression from the other sub-expressions. The remaining expression will replace the original expression, when the code of the `then` clause is going to be moved to the appropriate subclass.

To this end, the original expression is represented as a binary expression tree where all the parent nodes are conditional AND operators and the leaf nodes are the actual sub-expressions. Next, all the leaf nodes are examined (according to the first condition of the aforementioned rules) to identify a sub-expression that performs state-checking. If more than one sub-expressions are found to perform state-checking, then the technique selects the sub-expression whose state variable exists in the state-checking expressions of all the other `if` statements. The remaining expression is constructed by removing the identified leaf node from the binary tree and by replacing its parent node with its sibling node.

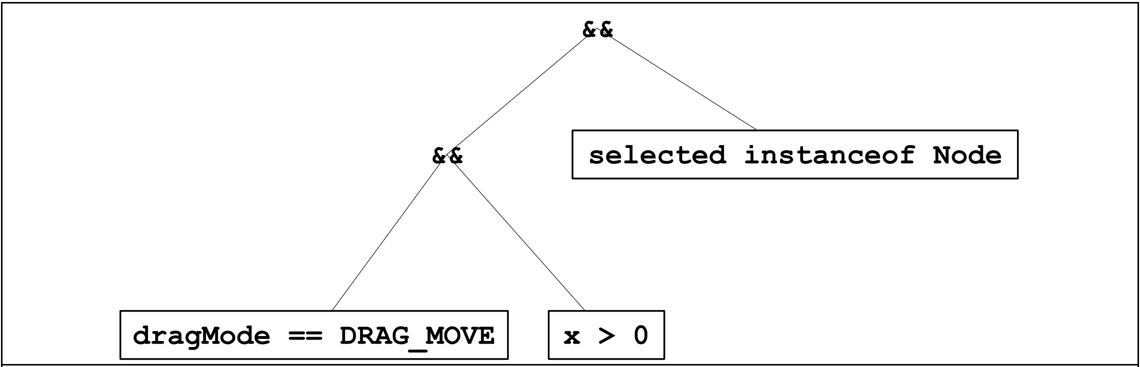
In the code example of Figure 4.2 the expression of the first `if` statement consists of three sub-expressions combined with conditional AND operators.

```
if (dragMode == DRAG_MOVE && x > 0 && selected instanceof Node) {
   ...
}
else if (dragMode == DRAG_LASSO) {
   ...
}
```

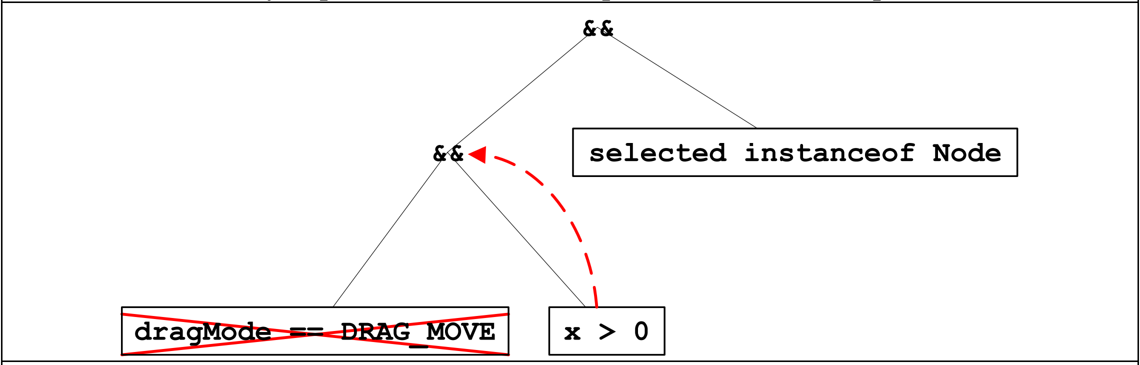
Figure 4.2: Example of compound conditional expression with AND operators.

The binary expression tree of the compound expression of Figure 4.2 is shown in Figure 4.3a. After examining the leaf nodes of the binary expression tree, two out of the three sub-expressions can be considered as valid state-checking expressions. The first one “`dragMode == DRAG_MOVE`” is an equality comparison of variable `dragMode` with the named constant `DRAG_MOVE`, while the third one “`selected instanceof Node`” is an `instanceof` expression used for the purpose of RunTime Type Identification. From the two valid state-checking expressions the first sub-expression is selected “`dragMode == DRAG_MOVE`”, since variable `dragMode` appears in the state-checking expression of the second `if` statement “`dragMode == DRAG_LASSO`”. Consequently, the remaining expression is constructed by removing leaf node “`dragMode == DRAG_MOVE`” from the tree and by replacing its parent node “`&&`” with its sibling node “`x > 0`”, as shown in Figure 4.3b. Finally, the remaining expression will be the compound conditional expression “`x > 0 && selected instanceof Node`”.

In the case where the expression of an `if` statement consists of sub-expressions combined with conditional OR operators (||) and all sub-expressions perform state-checking on the same variable holding the current state, the functionality of the `then` clause is common for all the named constants that participate in the state-checks. In order to avoid the duplication of the `then` clause in all the concrete subclasses corresponding to the named constants that participate in the state-checks, an intermediate class is introduced in the created State/Strategy inheritance hierarchy between the abstract class playing the role of State/Strategy and the corresponding concrete subclasses. The intermediate class overrides the polymorphic method of the State/Strategy superclass with the common functionality of the `then` clause, while the concrete subclasses simply inherit the intermediate class without overriding the polymorphic method. The name of the intermediate class is extracted from the corresponding named constants employing a Longest Common Subsequence (LCS) algorithm. In the code example of Figure 4.4a the expressions of both `if` statements consist of two sub-expressions combined with a conditional OR operator. Figure 4.4b shows the resulting State inheritance hierarchy which has two intermediate classes (`Destroy`, `Conquer`) containing the common functionality of the corresponding `if` statements.



a) Binary expression tree of a compound conditional expression

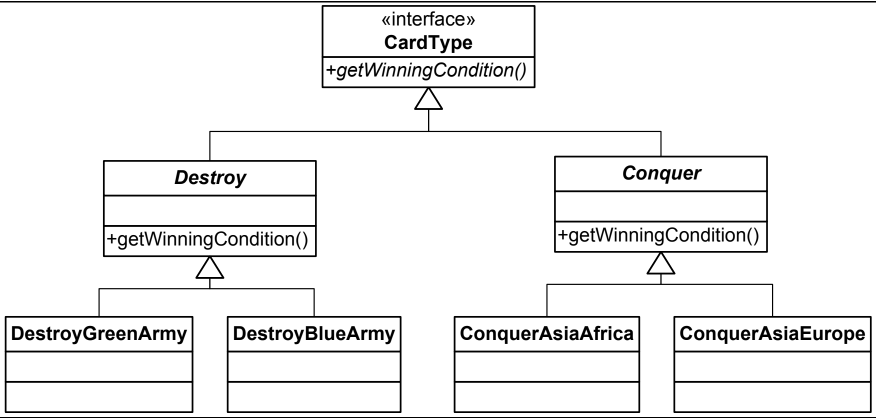


b) Construction of the remaining expression

Figure 4.3: Handling of compound conditional expressions with AND operators.

```
public boolean getWinningCondition() {
    if(cardType == DESTROY_GREEN_ARMY || cardType == DESTROY_BLUE_ARMY) {
        ...
    }
    else
    if(cardType == CONQUER_ASIA_AFRICA || cardType == CONQUER_ASIA_EUROPE) {
        ...
    }
}
```

a) Example of compound conditional expressions with OR operators



b) Resulting State hierarchy with an intermediate inheritance level

Figure 4.4: Handling of compound conditional expressions with OR operators.

4.3.4 Preconditions

According to Opdyke [86], each refactoring is associated with a set of preconditions which ensure that the behavior of a program will be preserved after the application of the refactoring. The proposed technique examines all valid cases of state-checking and disqualifies those cases that do not satisfy the following set of preconditions:

1. The state-checking code fragment should not contain assignments of local variables belonging to the LV_m set (variables of method *m* declared before the state-checking code fragment) or parameters belonging to P_m. In the case where such a variable is passed as parameter to the polymorphic method, its value would not change after the execution of the polymorphic method, since parameters are passed by-value in Java. This could possibly affect the behavior of the code that followed the state-checking code fragment after the application of the refactoring. An exception to this rule is the case where the state-checking code fragment contains assignments of a single variable belonging to the union of LV_m and P_m sets that is returned inside or after the state-checking code fragment.
2. The state-checking code fragment cannot be extracted if it resides inside the body of an iteration statement (`for`, `while`, `do-while`) and contains unstructured control flow statements (`break`, `continue`). The reason is that if a branch of the state-checking code is extracted as a separate method, then the contained unstructured control flow statement will not be surrounded by any iteration thus leading to a compilation error.
3. The state-checking code fragment should not contain any `super` method invocations, since the move of the code containing such invocations to the corresponding subclass would lead to compilation problems.
4. The names of the created classes belonging to the State/Strategy inheritance hierarchy should not be the same with the names of already existing classes in the same package or even in different packages. In the first case, it is not possible to have two classes with the same name in the same package. In the second case, the classes of the system that do not explicitly import the already existing class will present errors due to the ambiguous type of the conflicting class name. This issue can be resolved by renaming the classes of the State/Strategy hierarchy that lead to conflict.

4.3.5 Assessing the effect of the refactoring suggestions on design quality

It is reasonable to expect that several cases of state-checking may exist in a software system, especially when it consists of many classes. As a result, it is really important to be able to distinguish the refactoring suggestions which have greater effect on the design of the system. To this end, the proposed technique provides a sorting mechanism for the identified refactoring opportunities.

First of all, the refactoring suggestions are grouped according to their relevance. It can be considered that there are two kinds of grouping based on the nature of state-checking. The first one involves the cases that perform state-checking based on named constants and the grouping criterion is the named constants found in common. The second one involves the cases that perform RunTime Type Identification based on subclass types and the grouping criterion is the common inheritance hierarchy that the subclass types may belong to. The philosophy behind this kind of grouping is that the refactoring suggestions belonging to the same group will eventually utilize the same inheritance hierarchy (that either is going to be created or already exists). The groups of refactoring suggestions are sorted according to their size. The higher the number of the refactoring suggestions belonging to a group, the greater the impact of the specific group on design quality, since the degree of polymorphism (i.e., the number of polymorphic methods added to a single inheritance hierarchy) introduced to the system will be higher. In the case where two groups have the same size, they are sorted according to the average number of statements per branch (state) of the state-checking code fragments that they contain. Finally, the refactoring suggestions are also sorted within the group that they belong to according to the number of cases performing state-checking at each class of the group (at first level) and the average number of statements per branch of each state-checking code fragment in the group (at second level).

It should be noted that the refactoring suggestions within a group are independent with each other, in the sense that the application of a refactoring does not affect the other suggestions. In other words, the application of all refactorings belonging to a group leads to the same code regardless of the order in which they are applied.

4.3.6 Limitations

As already mentioned in the introduction, the proposed approach is semi-automatic in the sense that the decision of whether a refactoring suggestion should be accepted or not is left up to the designer of the examined program. Consequently, a limitation of the approach is that the effectiveness of the refactoring identification technique relies on the expertise of the designer. In general, the designer should consider three factors in order to derive a decision:

1. The number of conditional structures that perform state-checking on specific named constants throughout the code of the program. Obviously, the larger the number of these conditional structures, the more severe the design problem is. The proposed technique assists the designer by grouping the suggestions according to the relevance of the named constants participating in the conditional structures and sorting the resulting groups of suggestions according to their size.
2. The possibility of adding a new state to an already existing group of states due to a future change in requirements. Obviously, if the designer is absolutely sure that the addition of new states in the future is not possible, there is no need to replace the existing solution with one that introduces polymorphism. This factor can be determined based on the requirements of the program under examination.
3. The trade-off between the flexibility that may be introduced by the employment of State pattern and the complexity that may be caused by the number of concrete State subclasses being added. Obviously, the smaller the number of added State subclasses and the larger the size of code being moved to them, the more beneficial the refactoring is.

Assuming that the maintainer of the program under examination has a sufficient knowledge of its design structure and requirements, and exploits effectively the assistance provided by the proposed technique, the time required for the examination of the refactoring suggestions is significantly reduced.

The proposed technique does not cover all refactoring opportunities that introduce polymorphism. Kerievsky [54] proposed a catalogue of refactorings that replace simpler solutions to specific design problems with solutions introducing design patterns and consequently polymorphism. However, each design pattern requires a completely different approach for the identification of cases where it can be introduced. As a result, it is impossible to build a common technique that covers all refactoring opportunities introducing design patterns. It should be noted, though, that state-checking conditional logic has been widely recognized as an important design flaw in object-oriented software [36, 26, 54], since conditional logic is considered as one of the most common sources of complexity.

It should be noted that conditional structures performing RTTI can be refactored (in order to eliminate the conditionals by employing polymorphism) either by applying "Replace Conditional with Polymorphism" [36] or by applying "Move Accumulation to Visitor" [54]. In the first case the code contained in the conditional branches is moved to the subclasses of a common hierarchy, while in the second case the code is accumulated in a single Visitor class and double-dispatch is employed. In this particular context, the second approach (i.e., the Visitor design pattern) is required when the designer wants to avoid "polluting" the subclasses with additional operations [39]. For both cases the proposed technique would identify the need for introducing polymorphism; however, the selection between the two aforementioned solutions depends on factors that cannot be automatically determined. The tool currently automates only the application of the first solution.

Another issue deals with the existence of additional refactoring opportunities on the conditional structures performing state-checking. Usually, such conflicting refactorings involve the extraction of the code residing in conditional branches as new separate methods (this activity constitutes part of the *Refactor to Understand* reengineering pattern introduced by Demeyer et al. [26], the extraction of code that is duplicated between a conditional structure and other parts of the program as a single method, and even the move of a conditional structure (or the method containing it) to another class due to Feature Envy [36] design problem. The aforementioned refactorings can be considered as low-level transformations compared to more sophisticated refactorings introducing polymorphism and design patterns, and thus should be applied first. It should be noted that if the application of low-level refactorings does not affect the branching structure of conditional statements, then the refactoring opportunities introducing polymorphism will be preserved.

Finally, the conditional operator `?:` (also known as the *ternary operator*) has not been considered by the proposed technique. The ternary operator is used in the form of conditional expression

condition `?` *value_if_true* `:` *value_if_false*

where *value_if_true* is returned if *condition* is true, and *value_if_false* otherwise. This conditional expression is most commonly used as the right hand side of assignment statements. As a result, its usage potential is limited compared to `if` and `switch` statements. Moreover, it is not suitable for the representation of multiple cases or execution branches, since it results in overcomplicated code which is difficult to read and understand. For these reasons, the conditional expression with ternary operator is rarely used compared to `if` and `switch` statements.

4.3.7 Demonstration of the technique on an open-source project

This section presents the refactoring suggestions extracted by the proposed technique for an open-source project and demonstrates the application of two representative refactorings on source code. The examined open-source project is named Violet (version 0.16) and is a UML editor intended for students, teachers, and authors who need to produce simple UML diagrams quickly [47]. The extracted suggestions will be presented separately for the two kinds of refactoring solutions which are supported by the proposed technique.

The suggestions corresponding to Replace Type Code with State/Strategy refactorings are summarized in Table 4.1.

Table 4.1: Replace Type Code with State/Strategy refactoring suggestions for Violet (version 0.16).

Id	**Class**	**Method**	**Named constants**	**Default case**	**Name of variable holding the state**	**Kind of variable holding the state**	**#cases in a class utilizing the same hierarchy**	**#cases in a system utilizing the same hierarchy**	**Average #statements per branch**
1	GraphPanel	mouseDragged	DRAG_MOVE DRAG_LASSO	No	dragMode	field	3	3	16
2	GraphPanel	paintComponent	DRAG_RUBBERBAND DRAG_LASSO	No	dragMode	field			6.5
3	GraphPanel	mouseReleased	DRAG_RUBBERBAND DRAG_MOVE	No	dragMode	field			3.5
4	MultiLine-String	setLabelText	LEFT CENTER RIGHT	No	justification	field	1	1	1

* all class names are preceded by package "com.horstmann.violet.framework."

The group of refactoring suggestions 1-3 (Table 4.1) is related to named constants representing different drag modes. The active drag mode affects the way that UML components are painted in the diagram, as well as the handling of various mouse events. It can be considered that the State inheritance hierarchy created for this group of refactorings will be sufficiently utilized, since three polymorphic methods will be added (the number of polymorphic methods being added is equal to the size of the group) and a relatively large number of statements (as it is evident from the last column of Table 4.1) will be moved to the corresponding overriding methods in the concrete State subclasses when the refactorings of the group are applied. The application of Replace Type Code with State/Strategy refactoring for the third suggestion is demonstrated in Figure 4.5.

As it can be observed from Figure 4.5a, method `mouseReleased` in class `GraphPanel` contains an `if/else if` statement that performs state-checking. The state variable is instance variable `dragMode` having `int` type, while the set of identified named constants *INC* is {`DRAG_RUBBERBAND`, `DRAG_MOVE`} and the set of additional named constants *ANC* that results as described in Section 4.3.1 is {`DRAG_NONE`, DRAG_LASSO}. After the application of the refactoring, class GraphPanel plays the role of Context in the State/Strategy pattern, as shown in Figure 4.5b. The type of the state variable dragMode has been changed to the type of the abstract class DragMode playing the role of State/Strategy. The state-checking code fragment has been replaced with an invocation of the polymorphic method mouseReleased through state variable dragMode. Finally, each concrete State subclass (e.g., class DragMove that represents named constant DRAG_MOVE) overrides the polymorphic method mouseReleased by copying the statements of the corresponding conditional branches.

```java
public class GraphPanel extends JPanel {
   private Graph graph;
   private ToolBar toolBar;
   private double zoom;
   private Point2D mouseDownPoint;
   private int dragMode;

   private static final int DRAG_NONE = 0;
   private static final int DRAG_MOVE = 1;
   private static final int DRAG_RUBBERBAND = 2;
   private static final int DRAG_LASSO = 3;
   private static final int CONNECT_THRESHOLD = 8;

   public void mouseReleased(MouseEvent event) {
      Point2D mousePoint = new Point2D.Double(
         event.getX()/zoom, event.getY()/zoom);
      Object tool = toolBar.getSelectedTool();

      if(dragMode == DRAG_RUBBERBAND) {
         Edge prototype = (Edge)tool;
         Edge newEdge = (Edge)prototype.clone();
         if(mousePoint.distance(mouseDownPoint) >
               CONNECT_THRESHOLD
               && graph.connect(newEdge,
               mouseDownPoint, mousePoint)) {
            setModified(true);
            setSelectedItem(newEdge);
         }
      }
      else if(dragMode == DRAG_MOVE) {
         graph.layout();
         setModified(true);
      }

      dragMode = DRAG_NONE;
      revalidate();
      repaint();
   }
}
```

a) original code

```java
public class GraphPanel extends JPanel {
   private Graph graph;
   private ToolBar toolBar;
   private double zoom;
   private Point2D mouseDownPoint;
   private DragMode dragMode = new DragNone();

   public static final int DRAG_NONE = 0;
   public static final int DRAG_MOVE = 1;
   public static final int DRAG_RUBBERBAND = 2;
   public static final int DRAG_LASSO = 3;
   private static final int CONNECT_THRESHOLD = 8;

   public void mouseReleased(MouseEvent event) {
      Point2D mousePoint = new Point2D.Double(
         event.getX()/zoom, event.getY()/zoom);
      Object tool = toolBar.getSelectedTool();

      dragMode.mouseReleased(tool,mousePoint,this);

      setDragMode(DRAG_NONE);
      revalidate();
      repaint();
   }
}
```

Original class playing the role of Context

```java
public abstract class DragMode {
   public abstract int getDragMode();

   public abstract void mouseReleased(Object tool,
      Point2D mousePoint, GraphPanel graphPanel);
}
```

Abstract class playing the role of State

```java
public class DragMove extends DragMode {
   public int getDragMode() {
      return GraphPanel.DRAG_MOVE;
   }

   public void mouseReleased(Object tool,
      Point2D mousePoint, GraphPanel graphPanel) {

      graphPanel.getGraph().layout();
      graphPanel.setModified(true);
   }
}
```

Concrete State subclass representing *DRAG_MOVE*

b) refactored code

Figure 4.5: Application of Replace Type Code with State/Strategy refactoring.

The suggestions corresponding to Replace Conditional with Polymorphism refactorings are summarized in Table 4.2.

Table 4.2: Replace Conditional with Polymorphism refactoring suggestions for Violet (version 0.16).

Id	Class	Method	Subclass types	Default case	Name of superclass type reference	Kind of superclass type reference	#cases in a class utilizing the same hierarchy	#cases in a system utilizing the same hierarchy	Average #statements per branch
1	Sequence-Diagram-Graph	layout	CallNode ImplicitParameterNode	No	n	local variable	2	4	1
2	Sequence-Diagram-Graph	removeEdge	CallNode	Yes	end	getter invocation			1
3	CallEdge	getPoints	CallNode PointNode	Yes	n	local variable	1		5.3
4	Package-Node	addNode	ClassNode InterfaceNode PackageNode	Yes	n	parameter	1		1.5

* all class names are preceded by package “com.horstmann.violet.”

The group of refactoring suggestions 1-4 (Table 4.2) is related to subclass types that belong to the inheritance hierarchy of interface `Node`. The classes belonging to the `Node` inheritance hierarchy represent elements that participate in UML diagrams. The conditional structures corresponding to this group of suggestions perform Run-Time Type Identification based on the actual subclass type of the `Node` reference. The application of Replace Conditional with Polymorphism refactoring for the third suggestion is demonstrated in Figure 4.6.

As it can be observed from Figure 4.6a, method `getPoints` in class `CallEdge` contains an `if/else if` statement that performs RTTI. The reference to superclass type is local variable `n` whose type is `Node`. The set of identified subclass types *IST* is {`CallNode`, `PointNode`} and the conditional statement has also a final `else` clause (default implementation). The inheritance hierarchy tree structure corresponding to the identified subclass types is shown in Figure 4.7. The abstract class `RectangularNode` has eleven more subclasses which have not been included in the Class Diagram of Figure 4.7. After the application of the refactoring, the conditional code performing RTTI has been replaced with an invocation of the polymorphic method `getPoints` (declared in interface `Node`) through local variable `n`, as shown in Figure 4.6b. Each subclass belonging to *IST* overrides the polymorphic method `getPoints` by copying the statements of the corresponding conditional branches, while class `AbstractNode` provides the default implementation by copying the statements of the final `else` clause.

```java
public class CallEdge extends SegmentedLineEdge {
   public ArrayList getPoints() {
      ArrayList a = new ArrayList();
      Node n = getEnd();
      Rectangle2D start = getStart().getBounds();
      Rectangle2D end = n.getBounds();

      if (n instanceof CallNode &&
            ((CallNode)n).getImplicitParameter() ==
            ((CallNode)getStart()).
               getImplicitParameter()) {

         Point2D p = new Point2D.Double(
            start.getMaxX(),
            end.getY() - CallNode.CALL_YGAP / 2);
         Point2D q = new Point2D.Double(
            end.getMaxX(), end.getY());
         Point2D s = new Point2D.Double(
            q.getX() + end.getWidth(), q.getY());
         Point2D r = new Point2D.Double(
            s.getX(), p.getY());
         a.add(p);
         a.add(r);
         a.add(s);
         a.add(q);
      }
      else if (n instanceof PointNode) {
         a.add(new Point2D.Double(
            start.getMaxX(), start.getY()));
         a.add(new Point2D.Double(
            end.getX(), start.getY()));
      }
      else {
         Direction d = new Direction(
            start.getX() - end.getX(), 0);
         Point2D endPoint =
            getEnd().getConnectionPoint(d);
         if (start.getCenterX() < endPoint.getX())
            a.add(new Point2D.Double(
               start.getMaxX(), endPoint.getY()));
         else
            a.add(new Point2D.Double(
               start.getX(), endPoint.getY()));
         a.add(endPoint);
      }
      return a;
   }
}
```

a) original code

```java
public class CallEdge extends SegmentedLineEdge {
   public ArrayList getPoints() {
      ArrayList a = new ArrayList();
      Node n = getEnd();
      Rectangle2D start = getStart().getBounds();
      Rectangle2D end = n.getBounds();

      n.getPoints(a, start, end, this);
      return a;
   }
}
```

Original class after the replacement of conditional code with polymorphic method invocation

```java
public abstract class AbstractNode implements Node {
   public void getPoints(ArrayList a,
      Rectangle2D start, Rectangle2D end,
      CallEdge callEdge) {
      Direction d = new Direction(
         start.getX() - end.getX(), 0);
      Point2D endPoint =
         callEdge.getEnd().getConnectionPoint(d);
      if (start.getCenterX() < endPoint.getX())
         a.add(new Point2D.Double(
            start.getMaxX(), endPoint.getY()));
      else
         a.add(new Point2D.Double(
            start.getX(), endPoint.getY()));
      a.add(endPoint);
   }
}
```

Existing class **AbstractNode** providing the default implementation

```java
public class CallNode extends RectangularNode {
   public void getPoints(ArrayList a,
      Rectangle2D start, Rectangle2D end,
      CallEdge callEdge) {
      if (getImplicitParameter() ==
          ((CallNode)callEdge.getStart())
             .getImplicitParameter()) {
         Point2D p = new Point2D.Double(
            start.getMaxX(),
            end.getY() - CallNode.CALL_YGAP / 2);
         Point2D q = new Point2D.Double(
            end.getMaxX(), end.getY());
         Point2D s = new Point2D.Double(
            q.getX() + end.getWidth(), q.getY());
         Point2D r = new Point2D.Double(
            s.getX(), p.getY());
         a.add(p);
         a.add(r);
         a.add(s);
         a.add(q);
      }
   }
}
```

Existing class **CallNode** providing the functionality of the first branch

b) refactored code

Figure 4.6: Application of Replace Conditional with Polymorphism refactoring.

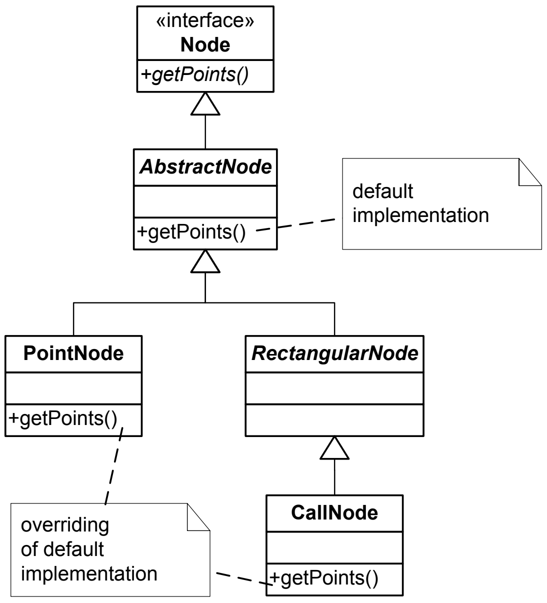
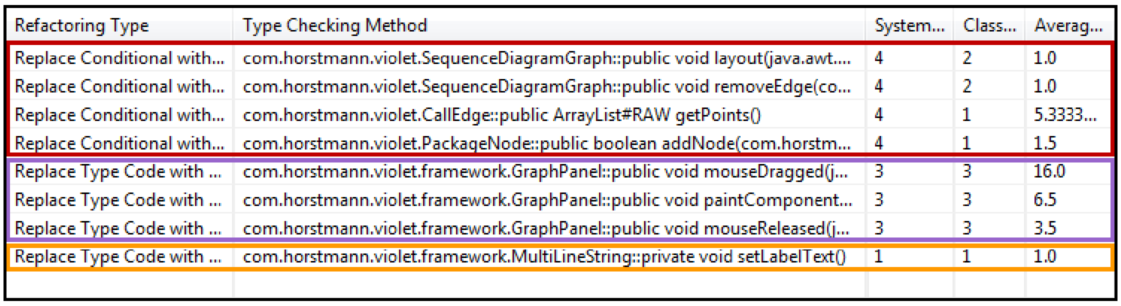


Figure 4.7: `Node` inheritance hierarchy tree structure.

4.4 JDeodorant Eclipse plug-in

The proposed technique has been implemented as an Eclipse plug-in [49] that not only identifies state-checking problems but also allows the user to apply the refactorings that resolve them on Java source code. Moreover, the tool groups the refactoring suggestions according to their relevance and sorts them within their groups according to the quantitative characteristics described in Section 4.3.5, assisting the user to determine an appropriate sequence of refactoring applications. The plug-in employs the ASTParser of Eclipse Java Development Tools (JDT) to analyze the source code of Java projects and the ASTRewrite to apply the refactorings and provide undo functionality. Figure 4.8 shows the way that the refactoring suggestions are presented to the user. The first column indicates the type of the extracted refactorings (*Replace Type Code with State/Strategy* or *Replace Conditional with Polymorphism*), while the second column indicates the method that contains the corresponding state-checking code fragment. By double-clicking on a row of the table the corresponding state-checking code fragment is highlighted in the Eclipse editor. The third and fourth columns indicate the number of relevant refactoring suggestions belonging to the same group at a system and class level, respectively. The final column shows the average number of statements per branch of the corresponding state-checking code fragment.



Refactoring Type	Type Checking Method	System...	Class...	Averag...
Replace Conditional with...	com.horstmann.violet.SequenceDiagramGraph::public void layout(java.awt....	4	2	1.0
Replace Conditional with...	com.horstmann.violet.SequenceDiagramGraph::public void removeEdge(co...	4	2	1.0
Replace Conditional with...	com.horstmann.violet.CallEdge::public ArrayList#RAW getPoints()	4	1	5.3333...
Replace Conditional with...	com.horstmann.violet.PackageNode::public boolean addNode(com.horstm...	4	1	1.5
Replace Type Code with ...	com.horstmann.violet.framework.GraphPanel::public void mouseDragged(j...	3	3	16.0
Replace Type Code with ...	com.horstmann.violet.framework.GraphPanel::public void paintComponent...	3	3	6.5
Replace Type Code with ...	com.horstmann.violet.framework.GraphPanel::public void mouseReleased(j...	3	3	3.5
Replace Type Code with ...	com.horstmann.violet.framework.MultiLineString::private void setLabelText()	1	1	1.0

Figure 4.8: Grouping and sorting of the refactoring suggestions.

4.5 Evaluation

The proposed technique has been evaluated in three ways:

a. The precision and recall of proposed technique is evaluated by performing an experiment to compare the refactoring opportunities identified by an independent expert to the results of the technique on various open-source projects.
b. The correlation of three quantitative factors (which are used to sort the refactoring suggestions extracted by the technique) with the decision of the independent expert to accept or reject the refactoring opportunities identified by the proposed technique is investigated.
c. The scalability of the proposed technique is evaluated by measuring the computation time required for the application of the technique with regard to the size of various open-source projects.

4.5.1 Evaluation of Precision and Recall

To evaluate the performance of the proposed technique in terms of exactness and completeness, an experimental study is performed in order to compare the findings of an independent expert to the results of the proposed technique on various open-source projects.

The expert that participated in the experiment had significant experience in software design (he has been working for more than 12 years as a telecommunications software designer) and deep knowledge of object-oriented design principles and patterns. Moreover, he was unfamiliar with the proposed technique and was able to dedicate a significant amount of time on analyzing the projects under study. The motivation behind his participation in the experiment was the utilization of the results for his PhD research on aspect-oriented design.

The projects which have been selected for the experiment are Violet 0.16 which is a UML editor intended for educational purposes, Ice Hockey Manager 0.1.1 which is a hockey team management game, and Nutch 0.4 which is a web crawler. The reasons for selecting these specific projects were:

- Their source code is open and publicly available allowing the replication of the experiment.
- They have a relatively small size allowing the independent expert to adequately examine them.
- They are implemented in Java programming language enabling the analysis of their source code by the proposed technique.
- The selected software releases correspond to rather immature versions, thus offering potential refactoring opportunities.
- They originate from different application domains allowing, to some extent, the generalization of the conclusions.

The size characteristics of the examined projects are shown in Table 4.3.

Table 4.3: Size characteristics of the examined open-source projects.

measure	Violet 0.16	IHM 0.1.1	Nutch 0.4
#classes	75	86	295
#methods with body	377	629	1389
total #conditional structures	231	245	1495
source lines of code	6910	8662	23579

* source lines of code (SLOC) have been measured using SLOCCount

The exact question that has been asked to the independent expert was: "Which are the conditional structures that should be replaced with an instance of State design pattern, or employ an RTTI mechanism that should be replaced with a polymorphic call?". True Occurrences are considered the refactoring opportunities reported by the independent expert. This set of True Occurrences is the baseline against which the proposed technique is compared when calculating precision and recall. The measures required for the classification of the results are defined as follows:

- True Positive: A refactoring opportunity identified by the independent expert, and also by the proposed technique.
- False Positive: A refactoring opportunity identified by the proposed technique, but not by the independent expert.
- False Negative: A refactoring opportunity identified by the independent expert, but not by the proposed technique.
- True Negative: A conditional structure which has not been considered to offer a refactoring opportunity by the independent expert and has not been suggested as a refactoring opportunity by the proposed technique.

The precision and recall for the examined projects are given in Table 4.4. It has been observed that when excluding from the suggestions of the proposed technique the conditional structures having an average number of statements per branch which is lower than two (i.e., conditional structures with a relatively small number of statements), the precision of the technique is significantly improved. The measures resulting from the application of the threshold (i.e., average number of statements per branch >= 2) are given inside parentheses wherever a change was observed.

Table 4.4: Precision and recall for the examined open-source projects.

Project	**Violet 0.16**		**IHM 0.1.1**		**Nutch 0.4**	
True Occurrences (TO)	7		9		17	
True Positives (TP)	4		9	(7)	17	
False Positives (FP)	4	(0)	12	(0)	13	(1)
False Negatives (FN)	3		0	(2)	0	
True Negatives (TN)	220	(224)	224	(236)	1465	(1477)
Precision: TP/(TP+FP)	50%	(100%)	43%	(100%)	57%	(94%)
Recall: TP/(TP+FN)	57%		100%	(78%)	100%	
Accuracy: (TP+TN)/(TP+FP+FN+TN)	97%	(99%)	95%	(99%)	99%	(100%)

The false negatives refer to conditional structures using `this` keyword in place of the variable holding the current state, whereas the proposed identification technique requires the existence of an instance variable, local variable, or method parameter. A conditional code that compares `this` keyword with a set of named constants (having the type of the class corresponding to `this` keyword) cannot be considered as a case of state-checking, since the value of `this` reference cannot change after object creation and as a result the state of the object cannot be modified at runtime. The independent expert supported that such cases could be eliminated by introducing a subclass (of the class corresponding to `this` reference) for each named constant that participates in the conditional and overriding the method that contains the conditional in each created subclass (i.e., by applying the Replace Type Code with Subclasses refactoring).

The independent expert reported a few cases where the variable holding the current state was a field inherited from a superclass. The proposed technique failed to collect these cases, because it requires the fields holding the current state to belong in the same class where the corresponding state-checking code fragment exists. The reason for this requirement is to make feasible the change of the original type of the field to the type of the abstract class playing the role of State. However, the expert supported that the original solutions should not be replaced with an instance of State pattern, and as a result, they were not considered as false negatives.

4.5.2 Correlation of quantitative factors with expert judgment

The goal of this experiment is to assess the correlation of three factors with the decision of the independent expert to accept or reject the refactoring opportunities identified by the proposed technique. These factors are:

a. The number of conditional structures performing state-checking on the same set of named constants or equivalently the number of polymorphic methods that can be added in the same inheritance hierarchy of states.

b. The number of alternative states belonging to a set of named constants or equivalently the number of concrete State subclasses that will be created in an inheritance hierarchy of states.
c. The average number of statements per branch in a conditional structure performing state-checking or equivalently the average number of statements that will be moved to the concrete State subclasses of an inheritance hierarchy.

The null and alternative hypotheses being tested are the following:

H_{0a} : The decision of accepting or rejecting a refactoring opportunity is not affected by factor a.

H_{1a} : The decision of accepting or rejecting a refactoring opportunity is affected by factor a.

H_{0b} : The decision of accepting or rejecting a refactoring opportunity is not affected by factor b.

H_{1b} : The decision of accepting or rejecting a refactoring opportunity is affected by factor b.

H_{0c} : The decision of accepting or rejecting a refactoring opportunity is not affected by factor c.

H_{1c} : The decision of accepting or rejecting a refactoring opportunity is affected by factor c.

The hypotheses will be tested by univariate logistic regression analyses, one for each factor. The dependent variable is a binary variable representing "agreement" or "disagreement" on the refactoring opportunities identified by the proposed technique, while the independent variable in each case is the corresponding factor. The values for the dependent variable were derived from the True Positives and False Positives of the experiment described in Section 4.5.1. A True Positive corresponds to an "agreement" of the independent expert with a refactoring opportunity identified by the proposed technique, while a False Positive corresponds to a "disagreement" of the independent expert with a refactoring opportunity identified by the proposed technique. The values for the independent variables (i.e., the three factors being examined in the experiment) were provided by the tool implementing the proposed technique. The data for the analysis have been drawn from project Nutch 0.4, since it provides the largest number of suggestions (N=30 cases). The results from the analysis are shown in Table 4.5.

Table 4.5: Logistic Regression Results for project Nutch (version 0.4).

Factor	B (S.E.)	Wald X^2	df	Nagelkerke R^2	p
a	1.777 (0.648)	7.533	1	0.783	0.006
b	-0.716 (0.254)	7.930	1	0.657	0.005
c	2.624 (0.920)	8.124	1	0.712	0.004

Since the p-value is less than the significance level (0.05) for all three factors, the null hypotheses can be rejected and claimed that the decision for accepting a refactoring opportunity is affected by all factors. In particular, considering the coefficients (B), the decision is affected positively by the number of polymorphic methods to be added to the same hierarchy of states (factor a) and the average number of statements that will be moved to the State subclasses (factor c), while it is affected negatively by the number of State subclasses that will be created (factor b). In other words, the independent expert tends to agree with the refactoring opportunities that sufficiently utilize a newly created inheritance hierarchy of states (by belonging to a relatively large group of opportunities that will utilize the same hierarchy of states), move a relatively large number of statements from the conditional branches to the corresponding State subclasses, and introduce hierarchies of states with a relatively small number of State subclasses. The proposed technique takes into account these quantitative factors when sorting the extracted refactoring suggestions in order to assist the designer in assessing their effect on design quality.

4.5.3 Threats to validity

Since the two experiments have different goals, their major threats are listed separately [110].

Threats to internal validity:

As threats to internal validity are considered those factors that may cause interferences regarding the relationships being investigated.

For the first experiment (Section 4.5.1), which is related with the evaluation of precision and recall of the technique, there is a possibility that the human expert has missed a number of refactoring opportunities while examining the code of the projects or misclassified a number of non-valid cases as refactoring opportunities. Obviously, these threats affect the reported precision and recall of the technique. The first threat is mitigated by the fact that the selected projects were relatively small in size and thus could be adequately examined by the independent expert. Moreover, the independent expert was motivated to perform a detailed and infallible analysis of the projects under study by the fact that the results would be utilized for his PhD research. The second threat is mitigated by the expertise of the evaluator and his past experience with design patterns in industrial software development.

For the second experiment (Section 4.5.2), which is related with the correlation of quantitative factors with expert judgment, there may have been omitted other important factors that affect the decision of the independent expert, such as the possibility of adding a new state to an already existing group of states due to a future change in requirements. Obviously, this threat could affect the accuracy of a multivariate prediction model which involves more than one independent variables as predictors at the same time, and for this reason, univariate regression analysis has been performed for each factor separately. In any case, the investigated statistical relationships do not prove a causal relationship between the factors and the expert's decision.

Threats to external validity:

Since the experiments have been performed employing a single expert as evaluator and a small number of projects, the study suffers from the usual threats to external validity. In other words, these factors limit the possibility to generalize the findings beyond the selected experimental settings (projects and evaluator). For example, in the experiment regarding the correlation of quantitative factors with expert judgment, the logistic regression results for the other two projects that have been considered in the first experiment (Violet 0.16 and IHM 0.1.1) were not statistically significant.

4.5.4 Evaluation of Scalability

The process required for the extraction of the refactoring suggestions in a given system consists of the following steps:

a. Parsing of the system under study using the ASTParser of Eclipse JDT.
b. Examination of all conditional statements (`switch`, `if/else if` statements) in the given system in order to identify valid cases of state-checking. Moreover, the valid cases of state-checking are checked against the set of preconditions defined at Section 4.3.4.

Table 4.6 contains various size measures for four open-source projects, namely Nutch 1.0, FreeCol 0.8.3, JMol 11.6.21, and JFreeChart 1.0.13. Table 4.7 presents the required computation time for each step of the process.

Table 4.6: Various size measures for the examined open-source projects.

measures	Nutch 1.0	FreeCol 0.8.3	JMol 11.6.21	JFreeChart 1.0.13
#classes	582	613	548	1037
#methods with body	2554	5104	6337	9960
#conditional structures	2391	5598	10730	8042
source lines of code	42955	83258	106237	143062

* source lines of code (SLOC) have been measured using SLOCCount

Table 4.7: CPU times for each step required for the extraction of refactoring suggestions.

step	Nutch 1.0	FreeCol 0.8.3	JMol 11.6.21	JFreeChart 1.0.13
a	7984 ms	17250 ms	13200 ms	20890 ms
b	200 ms	578 ms	1780 ms	734 ms

* Measurements performed on Intel Core 2 Duo E6600 2.4 GHz, 2 GB DDR2 RAM

The CPU time required for the first step depends on the size of the system under examination in terms of lines of code, since all field and method declarations (including the statements inside the body of each method) are parsed and analyzed. The CPU time required for the second step primarily depends on the total number of conditional structures found in the system under examination. The proposed technique requires access to the Abstract Syntax Tree (AST) information both during the identification of refactoring opportunities and the application of refactorings which are eventually selected by the user. A limitation regarding scalability is that the AST information of large projects occupies a large amount of heap memory causing OutOfMemory exceptions. This issue can be resolved either by applying the proposed technique on smaller components of a project (e.g., packages) or by removing AST information for classes that do not exhibit any refactoring opportunities.

Chapter 5

5 Identification of Extract Method Refactoring Opportunities

The extraction of a code fragment into a separate method is one of the most widely performed refactoring activities, since it allows the decomposition of large and complex methods and can be used in combination with other code transformations for fixing a variety of design problems. Despite the significance of Extract Method refactoring towards code quality improvement, there is limited support for the identification of code fragments with distinct functionality that could be extracted into new methods. The goal of the proposed approach is to automatically identify Extract Method refactoring opportunities which are related with the complete computation of a given variable (*complete computation slice*) and the statements affecting the state of a given object (*object state slice*). Moreover, a set of rules is proposed that exclude from being suggested as refactoring opportunities cases of slices whose extraction could possibly cause a change in program behavior.

5.1 Introduction

According to several empirical studies procedures/modules with large size [6], high complexity [41], and low cohesion [72] require significantly more time and effort for comprehension, debugging, testing and maintenance. A solution to this kind of design problems is given by Extract Method refactoring [36] which simplifies the code by breaking large methods into smaller ones and creates new methods which can be reused. However, existing IDEs and research approaches have focused on automating the extraction of statements which are indicated by the developer without providing support for the automatic identification of code fragments that could benefit from decomposition. Abadi et al. [1] stressed the inadequate support that is offered by modern IDEs for various cases requiring the application of Extract Method refactoring.

Extract Method refactoring is employed for fixing several design flaws such as *Duplicated Code* [36] where the same code structure existing in more than one places is extracted into a single method, *Long Method* [36] where parts of a large and complex method having a distinct functionality are extracted into new methods, *Feature Envy* [36] where a part of a method using several data of another class is initially extracted into a new method and then moved to the class that it envies. The wide use of Extract Method refactoring has been evident in several empirical studies [75, 76] that analyzed the refactoring operations performed by programmers using the Eclipse IDE.

The proposed approach covers the identification of refactoring opportunities which a) extract the complete computation of a given variable (referred to as *complete computation slice*) into a new method, b) extract the statements affecting the state of a given object (referred to as *object state slice*) into a new method. A complete computation slice is a slice that contains all the assignment statements of a given variable within the body of a method, while an object state slice is a slice that contains all the statements modifying the state of a given object (by method invocations through references pointing to this specific object) within the body of a method. It should be emphasized that object state slice has no relevance with the concept of *object slice* introduced by Liang and Harrold [65] which is defined as "*the statements in the methods of a particular object that might affect the slicing criterion*". Figure 5.1 illustrates two code examples for a complete computation slice and an object state slice, respectively.

The evaluation of the approach has shown that both complete computation and object state slices are able to capture code fragments implementing a distinct and independent functionality compared to the rest of the original method and thus lead to extracted methods with useful functionality.

```java
public void translate(double dx, double dy){
  if (getParent() == null) {
    dy = TOP_GAPY - getBounds().getY();
  }
  else {
    double y = getBounds().getY() + dy;
    y = Math.max(y,
        getParent().getBounds().getMinY()
        - topHeight / 2);
    y = Math.min(y,
        getParent().getBounds().getMaxY()
        - topHeight / 2);
    dy = y - getBounds().getY();
  }
  super.translate(dx, dy);
}
```

(a)

```java
public void draw(Graphics2D g2) {
  super.draw(g2);
  Color oldColor = g2.getColor();
  g2.setColor(color);

  Shape path = getShape();
  g2.fill(path);
  g2.setColor(oldColor);
  g2.draw(path);

  Rectangle2D bounds = getBounds();
  GeneralPath fold = new GeneralPath();
  fold.moveTo((float)(bounds.getMaxX()
    - FOLD_X), (float)bounds.getY());
  fold.lineTo((float)bounds.getMaxX()
    - FOLD_X, (float)bounds.getY() + FOLD_X);
  fold.lineTo((float)bounds.getMaxX(),
    (float)(bounds.getY() + FOLD_Y));
  fold.closePath();
  oldColor = g2.getColor();
  g2.setColor(g2.getBackground());
  g2.fill(fold);
  g2.setColor(oldColor);
  g2.draw(fold);

  text.draw(g2, getBounds());
}
```

(b)

Figure 5.1: (a) *complete computation slice* for variable `dy`. (b) *object state slice* for object reference `fold`.

The contribution of the proposed approach lies at the following points:

a. It introduces the concept of object state slice as a means to capture code that modifies the state of a given object and proposes an algorithm for the identification of such slices.
b. It proposes a set of rules that exclude from being suggested as refactoring opportunities cases of slices whose extraction could possibly cause a change in program behavior. Moreover, these rules cover the idiomorphic aspects of object-oriented programming languages.
c. It does not require any human intervention for the identification of refactoring opportunities. It can be regarded as semi-automatic in the sense that the designer will eventually decide whether an identified refactoring opportunity is beneficial or not.
d. It has been evaluated on a well-known open-source project.
e. It has been implemented as an Eclipse plug-in [49] allowing the reproduction of the results of the performed empirical study as well as its development by the software maintenance community.

5.2 Related Work

The vast majority of the papers found in the literature of function extraction are based on the concept of program slicing. According to Weiser [108], a slice consists of all the statements in a program that may affect the value of a variable x at a specific point of interest p. The pair (p, x) is referred to as *slicing criterion*. In general, slices are computed by finding sets of directly or indirectly relevant statements based on control and data dependences. After the original definition by Weiser, several notions of slicing have been proposed. Concerning the employment of runtime information, *static* slicing uses only statically available information to compute slices, while *dynamic* slicing [57] uses as input the values of variables for a specific execution of a program in order to provide more accurate slices. Concerning flow direction, in *backward* slicing a slice contains all statements and control predicates that may affect a variable at a given point, while in *forward* slicing [9] a slice contains all statements and control predicates that may be affected by a variable at a given point. Concerning syntax preservation, *syntax-preserving* slicing simplifies a program only by deleting statements and predicates that do not affect a computation of interest, while *amorphous* slicing [43] employs a range of syntactic transformations in order to simplify the resulting code. Concerning slicing scope, *intraprocedural* slicing computes slices within a single procedure, while *interprocedural* slicing [48] generates slices that cross the boundaries of procedure calls. Program slicing has several applications in various software engineering domains such as debugging, program comprehension, testing, cohesion measurement, maintenance and reverse engineering [101, 10, 42].

A direct application of program slicing in the field of refactorings is *slice extraction*, which has been formally defined by Ettinger [34] as the extraction of the computation of a set of variables V from a program S as a reusable program entity, and the update of the original program S to reuse the extracted slice. Within the context of slice extraction the literature can be divided into two main categories according to Ettinger [34]. In the first category belong the methodologies that extract slices based on a set of selected statements which are indicated by the user (*arbitrary method extraction*). In the second category belong the methodologies that extract slices based on a variable of interest at a specific program point which is indicated by the user.

The first approach for decomposing a procedure was proposed by Gallagher and Lyle [38]. They introduce the concept of *decomposition slice* as a slice that captures all computation on a given variable. The decomposition slice for a variable v is the union of the slices that result by using as seed statements in slicing criteria the statements that output variable v along with the last statement of the procedure. As output statement is considered a statement that prints or returns the value of a given variable. They also defined dependence relations between the resulting decomposition slices of a procedure. Two decomposition slices $S(v)$ and $S(w)$ are considered as *independent* if their intersection is empty ($S(v) \cap S(w) = \varnothing$). Decomposition slice $S(v)$ is considered as *strongly dependent* on $S(w)$ if $S(v)$ is a proper subset of $S(w)$ ($S(v) \subset S(w)$). The dependence relationships between the decomposition slices are used to construct the *lattice* of decomposition slices, which can be considered as a directed graph where nodes represent the decomposition slices of a procedure and edges represent the strongly dependent relationships between them. Figure 5.2b shows the lattice of decomposition slices for the code in Figure 5.2a. The decomposition slices for the output variables of the code in Figure 5.2a are the following: $S(c) = \{12, 13, 24\}$, $S(nc) = \{11, 12, 13, 14,$

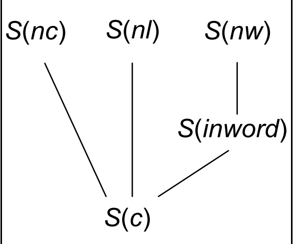
24}, *S*(*nl*) = {9, 12, 13, 15, 17, 18, 24}, *S*(*inword*) = {8, 12, 13, 15, 16, 20, 21, 24}, and *S*(*nw*) = {8, 10, 12, 13, 15, 16, 20, 21, 22, 24}.

As it can be observed from Figure 5.2b, *S*(*c*) is strongly dependent on all other decomposition slices and *S*(*inword*) is strongly dependent on *S*(*nw*). Tonella [102] introduced the *concept lattice* of decomposition slices as an extension to decomposition slice graph [38] in order to represent *weak inferences* (i.e., shared statements which are not decomposition slices) between decomposition slices. For example, statement 15 in Figure 5.2a is shared by decomposition slices *S*(*nl*), *S*(*inword*) and *S*(*nw*) but does not form a decomposition slice. Figure 5.2c illustrates the concept lattice of decomposition slices for the code in Figure 5.2a.

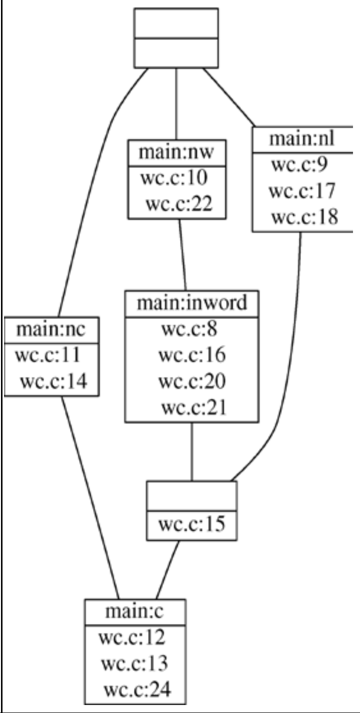
By examining Figure 5.2c, it can be observed that by traversing the concept lattice from the bottom to the decomposition slice of a variable *v* whose computation is intended to be extracted, the slice of variable *v* is the union of the statements in the traversed decomposition slices, while the statements that will be duplicated if the slice is extracted is the union of the statements in the traversed decomposition slices excluding the decomposition slice of variable *v*. The lattice of decomposition slices is used by Gallagher and Lyle [38] in order to construct the *complement* of a decomposition slice (i.e., the statements that should remain in the original procedure after the extraction of the decomposition slice). The proposed approach in a similar manner computes the *indispensable* statements corresponding to a slice. The indispensable statements are statements that although belong to the slice, should not be removed from the original method due to the existence of remaining statements in the original method (i.e., statements not belonging to the slice) which are dependent on them.

```
1   #define EOF -1
2   #define NO 0
3   #define YES 1
4
5   main() {
6     int inword, nl, nw, nc, c;
7
8     inword = NO;
9     nl = 0;
10    nw = 0;
11    nc = 0;
12    c = getchar();
13    while (c != EOF) {
14      nc = nc + 1;
15      if (c == ' ' || c == '\n') {
16        inword = NO;
17        if (c == '\n')
18           nl = nl + 1;
19      }
20      else if (inword == NO) {
21        inword = YES;
22        nw = nw + 1;
23      }
24      c = getchar();
25    }
26    printf("%d \n", nl);
27    printf("%d \n", nw);
28    printf("%d \n", nc);
29  }
```

(a)



(b)



(c)

Figure 5.2: (a) The code of a word counting program. (b) The lattice of decompositions slices according to Gallagher and Lyle [38]. (c) The concept lattice of decompositions slices according to Tonella [102].

The major difference of the proposed approach with decomposition slicing is related with the selection of the seed statements which are required to derive the computation of a given variable. The decomposition slicing technique uses as seed statements the statements that output the variable under consideration along with the last statement of the procedure. However, the selected seed statements may include code which is irrelevant with the computation of the variable under consideration, or even lead to the inclusion of additional irrelevant statements in the resulting slices due to the use of multiple variables within the seed statements. On the other hand, the proposed approach uses as seed statements the statements where the variable under consideration is defined, leading to slices that contain the pure variable computation.

Cimitile et al. [23] proposed a specification driven slicing process for identifying reusable functions based on the precondition and postcondition of a given function. Initially, a symbolic execution technique is used to recover the preconditions for the execution of each statement and predicate existing within the body of the function. Eventually, the statements whose preconditions are equivalent to the pre and post conditions of the function serve as candidate entry and exit points of the computed slice (i.e., a pair of statements restricting the expansion of the slice within their boundaries). A limitation of the approach is that human intervention is required to associate the data of the function's specification with the program variables, to define the set of output variables of the function and to provide invariant assertions that cannot be automatically derived.

Lanubile and Visaggio [62] introduced the notion of *transform* slicing as a method for extracting reusable functions. A transform slice includes the statements which contribute directly or indirectly to transform a set of input variables into a set of output variables. The computation of a transform slice is similar to the computation of a static backward slice with the difference that it expands until the statements that define values for the input variables are included in the slice. Transform slicing uses output statements as seeds for the slicing criteria, or the last program statement if it is not possible to find an output statement in the proper place. A limitation of the approach is that it presupposes the knowledge that a function is performed in the examined code and its specification in terms of input and output data. This kind of information can be provided only by means of human interaction. Moreover, the underlying method has not been empirically evaluated.

Kang and Bieman [51] proposed the *input-output dependence graph* (IODG) as a means to model and visualize the dependency relationships between inputs and outputs of a module. Based on the IODG representation of a module they defined the design-level cohesion (DLC) measure which provides an objective criterion for evaluating and comparing alternative design structures. Moreover, the DLC measure can be used as a criterion to determine whether or not a given module should be redesigned or restructured. Based on the IODG representation and the DLC measure they defined eight basic restructuring operations (i.e., module decomposition and composition operations) and described a process for applying the restructuring operations to improve design of system modules. The main limitation of the approach is that the restructuring process requires human intervention in order to specify expected marginal DLC levels of the examined modules, to decompose the IODG of the poorly designed modules in appropriate partitions exhibiting higher DLC level, and to locate unnecessarily decomposed modules based on the IODG visualization and aided by coupling, size, and/or reuse measures.

Lakhotia and Deprez [60] proposed a transformation, called *Tuck*, which can be used to restructure a program by breaking its large functions into smaller ones. The tuck transformation consists of three steps: Wedge, Split, and Fold. The wedge is a program slice that contains all the statements that influence a given set of seed statements. The split transformation splits the original function into two single-entry, single-exit (SESE) regions, one containing all the computations relevant to the set of seed statements and the other containing all the remaining computations. The transformation introduces new variables or renames variables and composes the two new regions so that the overall computation remains unchanged. Finally, the fold transformation creates a function for the SESE region corresponding to the seed statements and replaces the statements by a call to this function. A major limitation of the approach is that the tuck transformation requires as external input a set of seed statements, and a foldable subgraph (i.e., a subgraph where there is no edge from its exit node to any node of the subgraph) containing the seed statements. Furthermore, the evaluation performed by Komondoor and Horwitz [56] has shown that the performance of the approach was poor on a dataset of "difficult" cases because it does not allow the duplication of predicates, promotes statements in a non-intelligent manner (i.e., copies/moves unnecessary code to the extracted function) and does not handle exiting jumps.

Komondoor and Horwitz [56] proposed a method that takes as input the control flow graph of a procedure and a set of statements to be extracted (marked statements) and applies semantics-preserving transformations to make the marked statements form a contiguous, well-structured block that is suitable for extraction. The applied transformations are the reordering of unmarked statements in order to make the marked statements contiguous, the duplication of predicates in both the extracted and original procedure, the promotion of unmarked statements to the marked ones, and the special handling of exiting jumps such as return, break and continue statements. The first disadvantage of the approach is that it requires external intervention for the selection of the initial marked statements (which can be performed by the programmer or program analysis tools). The second disadvantage of the approach is that it does not allow the duplication of assignment statements and loop predicates leading to missed extraction opportunities in favor of low code duplication.

Harman et al. [44] introduced a variation of the algorithm proposed by Komondoor and Horwitz [56] which is based on amorphous procedure extraction. Amorphous extraction relaxes the syntactic constraints of the original program in order to enable the application of simplifying transformations. However, it retains the requirement that the extracted program and the original must be semantically equivalent. The goal of the proposed variation is to minimize the need for statement promotion (i.e., when a statement which was not originally marked for extraction must be extracted to preserve the semantics of the program) and predicate duplication in order to make the extraction process more precise. The main limitation of the approach is that it requires the identification of marked statements (i.e., the statements whose extraction is desired) to be performed by an external identification tool or a software maintainer.

Jiang et al. [50] performed an empirical study on six open-source projects in order to evaluate the splitability of procedures. Concerning the frequency of splitable procedures, they concluded that the majority of procedures are not splitable, while those which are splitable can be split into two or three subprocedures. Furthermore, they studied the *overlap* distribution of splitable procedures. Overlap is a measure of code duplication between the resulting subprocedures. The higher the overlap, the more cohesive the original procedure is, and therefore, less likely to be splitable. They concluded that the splitability of a procedure depends on the inter-dependency between its subprocedures. The higher the inter-dependency of subprocedures, the more statements they share with each other, and splitting generates a larger amount of duplicated code. Finally, the empirical results have shown a strong correlation between procedure size and splitability in the case of 2-way splitable procedures.

The aforementioned methods apart from requiring significant external input in terms of seed statements or input/output variables in order to operate, concern only procedural programming languages. Therefore, they do not take into account important issues regarding various aspects of object-oriented programming languages.

Maruyama [70] simplified an interprocedural slicing algorithm proposed by Larsen and Harrold [63] by making it intraprocedural and then introduced the concept of block-based region into the resulting algorithm. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. Maruyama employed a block-partitioning algorithm in order to decompose the control flow graph of a method into basic blocks and form several block-based regions used for restricting the expansion of slice within their boundaries. In this way it is possible to extract more than one slices for a given slicing criterion by using the appropriate block-based regions, compared to classic static slicing algorithms that extract only a single slice for a given slicing criterion by using the whole source method as target region. Although the approach of Maruyama was the first to cover slice extraction in object-oriented programming languages, it suffers from several limitations. It requires the indication of a variable of interest at a specific program point by the programmer in order to operate. It does not handle behavior preservation issues that can be raised from duplication of statements. It does not guarantee that the complete computation of the variable indicated by the user will be extracted as a separate method. Finally, it does not support extraction opportunities which are related with objects but only with variables of primitive type.

5.3 Method

The proposed method handles two main categories of Extract Method refactoring opportunities. The first category refers to variables (having primitive data types or being object references) whose value is modified by assignment statements throughout the body of the original method. The second category refers to object references (which are local variables or fields of the class containing the original method) pointing to objects whose state is affected by method invocations throughout the body of the original method. It should be noted that the state of an object reference is affected by method invocations that modify the value of at least one of its attributes. In the first case, the goal is to extract the complete computation of a given variable (*complete computation slice*), while in the second case, the goal is to extract all the statements modifying the state of a given object (*object state slice*) within the scope of the original method. The aforementioned goals ensure at a certain degree that the extracted code will exhibit useful functionality. To achieve these goals the proposed approach employs the union of static slices by different means according to the specific needs of each category. According to De Lucia et al. [25] the unions of static slices which rely on slicing algorithms that do preserve a subset of the direct data and control dependence relations of the original program are valid slices.

5.3.1 Construction of the Program Dependence Graph

The proposed approach employs the *Program Dependence Graph* (PDG) in order to represent the methods under examination. The Program Dependence Graph was initially introduced by Ferrante et al. [35] in order to represent control and data flow dependences between the operations of a procedure. The nodes of a PDG represent the statements of the corresponding procedure. Each node has a set of *defined* variables which consists of the variables whose value is modified by an assignment, and a set of *used* variables which consists of the variables whose value is used at the corresponding statement. A control dependence edge from node p to node q denotes that the execution of statement q depends on the control conditions of statement p. The sets of defined and used variables are employed to compute data dependences between the statements throughout the procedure control flow. A data dependence edge from node p to node q due to variable x denotes that statement p defines variable x, statement q uses variable x and there exists a control flow path from statement p to q without an intervening definition of x.

Later on, Horwitz et al. [48] introduced the *System Dependence Graph* (SDG) in order to represent procedure calls between PDGs and face the problem of interprocedural slicing (i.e., slicing that crosses the boundaries of procedure calls). A procedure call is represented using a *call-site* vertex, while the information transfer is represented using four kinds of *parameter* vertices, namely *actual-in* and *actual-out* on the calling side (representing assignment statements that copy the values of the actual parameters to the call temporaries and from the return temporaries, respectively) and *formal-in* and *formal-out* in the called procedure (representing assignment statements that copy the values of the formal parameters from the call temporaries and to the return temporaries, respectively). The PDGs are connected using three kinds of edges: a *call* edge is added from each call-site vertex to the corresponding procedure-entry vertex, a *parameter-in* edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure, and a *parameter-out* edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

Larsen and Harrold [63] extended the *System Dependence Graph* (SDG) proposed by Horwitz et al. [48] to represent object-oriented programs. They introduced the *Class Dependence Graph* (ClDG) to represent the methods and instance variables belonging to a class. Additionally, they proposed ways to represent inherited methods, class instantiations and polymorphic method calls. Liang and Harrold [65] improved the aforementioned approach by providing a way to distinguish data members for different objects instantiated from the same class.

Since the proposed approach aims at extracting intraprocedural slices (i.e., slices that extend within the boundaries of a method) as new separate methods, the PDG representation has been adopted which does not include any method call representation elements. However, the information regarding the state of the objects being referenced inside the body of a method is crucial for the formation of precise and correct slices, as well as the preservation of program behavior after code extraction. The state of an object reference can be modified or accessed by invoked methods which modify or access the fields of the object inside their body. These methods can be invoked directly by using the object reference as invoker, or indirectly by passing the object reference as parameter to another method which in turn uses the object reference as invoker.

Let us assume that statement *s* inside the body of method *m* invokes a method through object reference *r* or passes object reference *r* as parameter to a method. A partial call graph is recursively generated starting from method *m* that includes only the method calls which are associated with object reference *r* (i.e., methods which are actually invoked through the original reference *r* or the original reference *r* is passed as parameter to them). While the partial call graph is constructed, the fields which are modified or used inside the body of each visited method are added to the sets of defined and used variables of statement *s*, respectively. These fields are represented as composite variables (i.e., variables consisting of more than one parts), where the last part is the name of the corresponding field and the initial part is the actual reference through which the field was modified or accessed.

In the code example of Figure 5.3, statement 5 of method `main` invokes method `addRental` through object reference `customer` and passes as parameter to the invoked method object reference `rental`. The partial call graph corresponding to this method invocation is shown in Figure 5.4. At each method node in the call graph the sets of defined and used variables are shown, where the formal parameters have been replaced with the actual parameters (e.g., in method `addElement` of class `Vector`, formal parameter `obj` has been replaced with actual parameter `rental`) and `this` reference has been replaced with the actual invoker reference (e.g., in method `addRental` of class `Customer`, `this` reference has been replaced with the actual invoker reference `customer`). The sets of defined and used variables for statement 5 are actually the unions of the defined and used variable subsets, respectively, for each method in the call graph.

The computation of data dependences in the PDG of method *m* takes also into account the composite variables which are related with the state of object references existing in the body of *m*. These additional data dependences allow the formation of more precise and correct slices and at the same time enable the extraction of code that affects the state of a given object reference.

```
public class Customer {
  private String _name;
  private Vector _rentals = new Vector();

  public Customer (String name) {
    _name = name;
  }
  public void addRental(Rental arg) {
    _rentals.addElement(arg);
  }
1 public static void main(String args[]) {
2   Customer customer =
        new Customer("customer");
3   Movie movie =
        new Movie("title",Movie.NEW_RELEASE);
4   Rental rental = new Rental(movie, 3);
5   customer.addRental(rental);
  }
}
```

```
public class Vector<E> extends AbstractList<E> {
  protected Object[] elementData;
  protected int elementCount;
  protected int capacityIncrement;

  public synchronized void addElement(E obj) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = obj;
  }
  private void ensureCapacityHelper(int minCapacity){
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
      Object[] oldData = elementData;
      int newCapacity = (capacityIncrement > 0)
        ? (oldCapacity + capacityIncrement)
        : (oldCapacity * 2);
      if (newCapacity < minCapacity) {
        newCapacity = minCapacity;
      }
      elementData =
        Arrays.copyOf(elementData, newCapacity);
    }
  }
}
```

Figure 5.3: Code example to demonstrate the handling of method invocations.

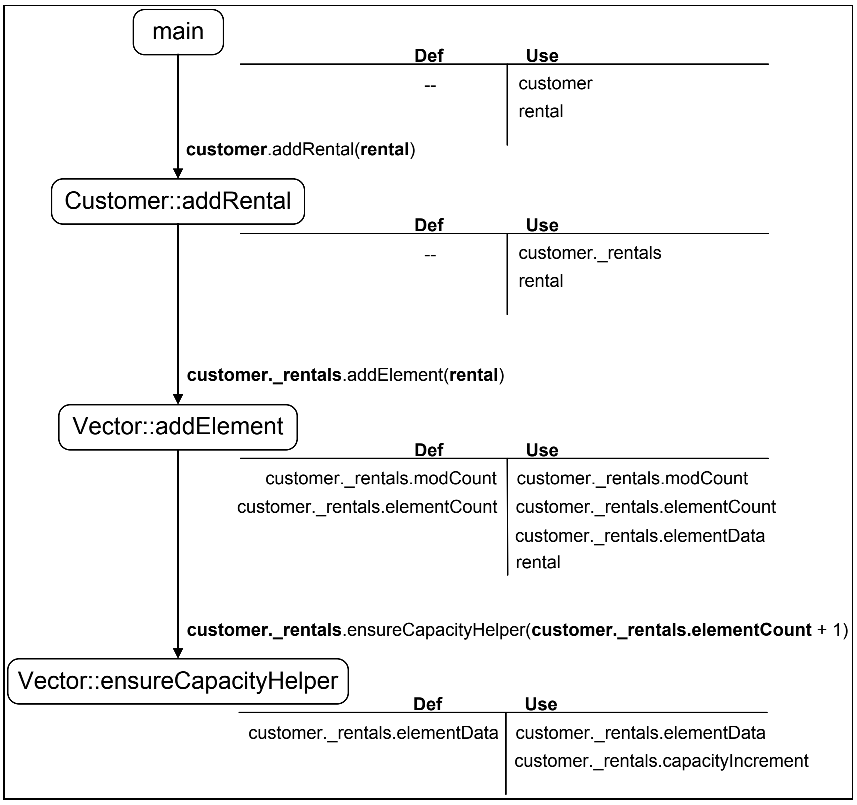

main
Def
Use
--
customer
rental
customer.addRental(rental)
Customer::addRental
Def
Use
--
customer._rentals
rental
customer._rentals.addElement(rental)
Vector::addElement
Def
Use
customer._rentals.modCount
customer._rentals.elementCount
customer._rentals.modCount
customer._rentals.elementCount
customer._rentals.elementData
rental
customer._rentals.ensureCapacityHelper(customer._rentals.elementCount + 1)
Vector::ensureCapacityHelper
Def
Use
customer._rentals.elementData
customer._rentals.elementData
customer._rentals.capacityIncrement

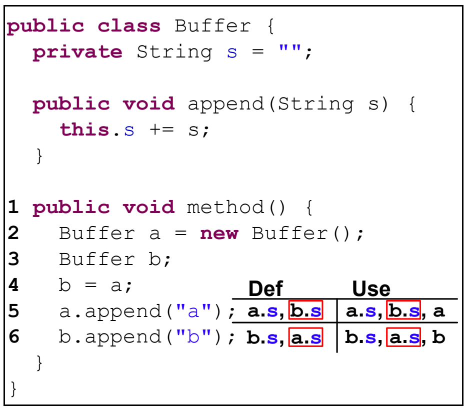
Figure 5.4: Call graph for statement 5 of method `main` in Figure 5.3, along with the sets of defined and used variables for each visited method (the actual references through which the methods are invoked and parameters are highlighted in bold).

The proposed approach adopted a variety of code analysis techniques in order to further increase the precision and correctness of the resulting slices.

a. *Alias analysis* [85]: An *alias relationship* exists between two references when they refer to the same object in memory during program execution. The set of references in which each element pair satisfies an alias relationship is called an *alias set*. Alias analysis is a method for extracting alias sets by static code analysis. Alias analysis techniques are mainly divided into two categories, namely *flow insensitive* where the execution order of statement is not taken into account and *flow sensitive* where the execution order of statements is taken into account. Flow sensitive techniques follow the control flow of a program in order to determine alias relationships and as a result they can extract more accurate alias relations compared to flow insensitive approaches. Landi et al. [61] have introduced the concept of *reaching alias sets* in order to compute flow sensitive alias relationships. A reaching alias set for a given statement is a collection of alias sets which apply just before the execution of this statement. For example, in the code of Figure 5.5 the reaching alias set for both statements 5 and 6 is ⟨`a`, `b`⟩, since after the execution of statement 4 references `a` and `b` point to the same object in memory. The proposed approach handles the existence of a reaching alias set *RASet* for statement *s* in the following way:

   For each composite variable in the sets of defined and used variables of statement *s* whose first part is a reference *r* belonging to an alias set *A* of *RASet*, an additional number of composite variables is added (to the set of defined or used variables, respectively) which is equal to the number of references belonging to alias set *A* (excluding *r*) by replacing the first part of the composite variable with each one of the aliases of reference *r*.

In the example of Figure 5.5, the additional composite variables that were added in the sets of defined and used variables are highlighted in rectangles. In this way, it is ensured that in the case of an alias relationship all statements affecting the state of the same object in memory will be extracted together regardless of the actual references through which the methods changing the object's state are invoked.



```
public class Buffer {
  private String s = "";

  public void append(String s) {
    this.s += s;
  }

  public void method() {
    Buffer a = new Buffer();
    Buffer b;
    b = a;                 Def        Use
    a.append("a"); a.s, b.s  | a.s, b.s, a
    b.append("b"); b.s, a.s  | b.s, a.s, b
  }
}
```

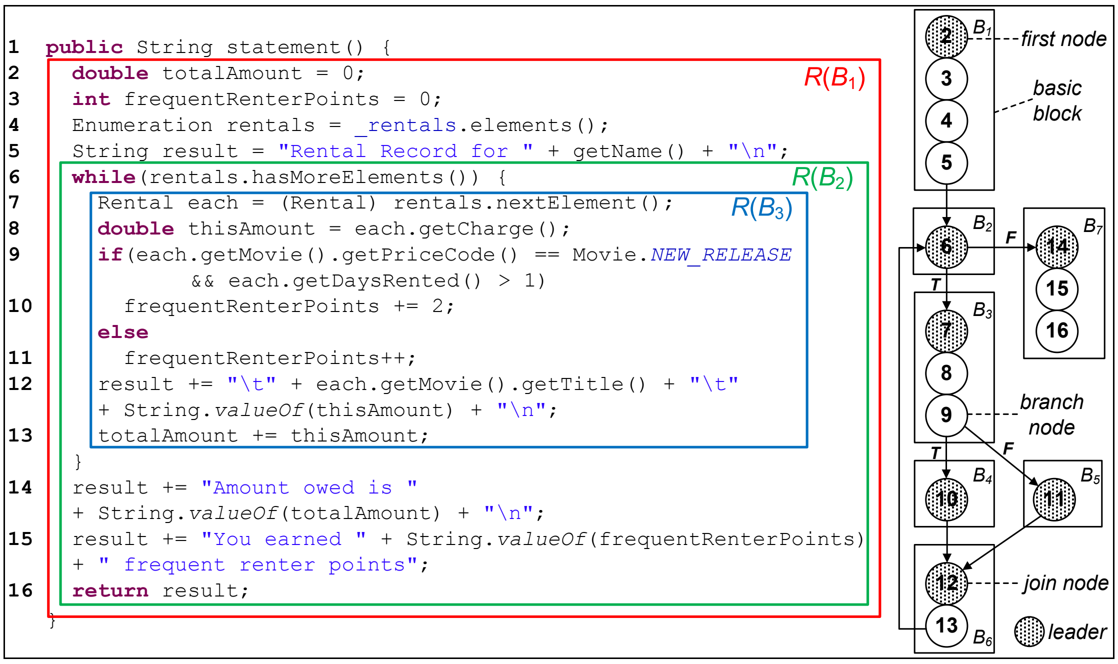
Figure 5.5: Code example containing an alias relationship between references a and b (the composite variables that were added in the sets of defined and used variables due to the existence of alias set ⟨a, b⟩ are highlighted in rectangles).

b. *Polymorphic method call analysis* [63, 65]: A polymorphic method call occurs when an abstract method is invoked through a reference of abstract type. Usually, the actual subclass type of the reference can be determined only at runtime. When the type of the caller reference cannot be statically determined, all concrete implementations of the abstract method are visited in the respective call graph. In this way, it is ensured that the state information associated with the caller reference covers all possible subclass types that the reference may obtain at runtime.

5.3.2 Block-based Slicing

Traditional intraprocedural slicing algorithms use the entire method body as a region where the slice may expand starting from the statement of the slicing criterion. However, within the context of slice extraction, where the goal is to extract the resulting slice as a new separate method, the extraction of a slice that expands throughout the entire method body is not always feasible. Maruyama introduced the concept of block-based slicing [70] as a means for producing more than one slices for a given slicing criterion. This is achieved by constructing block-based regions within the body of a method, which can be used to restrict the expansion of a slice within their boundaries. In the proposed approach, block-based slicing helps to determine regions of the original method where slices starting from statements that belong to different blocks and concern the computation of the same variable can be extracted together as a union. The block-based regions of method *m* can be determined in the following way:

As a first step, the control flow graph of method *m* is constructed in order to decompose it into basic blocks. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. A block-partitioning algorithm [3] marks as *leader* nodes the first node, the join nodes (i.e., the nodes which have two or more incoming flow edges), and the nodes that immediately follow a branch node (i.e., a node which has two or more outgoing flow edges) in the control flow graph of the method. For each leader node, its basic block consists of itself and all subsequent nodes up to the next leader or the last node in the control flow graph. Figure 5.6 illustrates the control flow graph (decomposed into basic blocks) for method `statement()` used in a well-established refactoring example [36].



```
1  public String statement() {
2    double totalAmount = 0;
3    int frequentRenterPoints = 0;
4    Enumeration rentals = _rentals.elements();
5    String result = "Rental Record for " + getName() + "\n";
6    while(rentals.hasMoreElements()) {
7      Rental each = (Rental) rentals.nextElement();
8      double thisAmount = each.getCharge();
9      if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE
              && each.getDaysRented() > 1)
10       frequentRenterPoints += 2;
       else
11       frequentRenterPoints++;
12     result += "\t" + each.getMovie().getTitle() + "\t"
       + String.valueOf(thisAmount) + "\n";
13     totalAmount += thisAmount;
     }
14   result += "Amount owed is "
     + String.valueOf(totalAmount) + "\n";
15   result += "You earned " + String.valueOf(frequentRenterPoints)
     + " frequent renter points";
16   return result;
   }
```

Figure 5.6: Method `statement()` and its corresponding control flow graph.

Maruyama defined as *reachable blocks* (*Reach*(*B*)) for basic block *B*, the set of blocks that can be reached from *B* on the control flow graph without traversing loop-back edges. For example, the reachable blocks for basic block B_3 in the control flow graph of Figure 5.6 is set $Reach(B_3) = \{B_3, B_4, B_5, B_6\}$, since the loopback edge from statement 13 to statement 6 is excluded from being traversed.

Based on the definition of reachable blocks, Maruyama defined as *block-based region* $R(B_n)$ for boundary block B_n the set of nodes which belong to $Reach(B_n)$. Figure 5.6 depicts the statements that belong to regions $R(B_1)$, $R(B_2)$ and $R(B_3)$, respectively. In terms of program dependency, a block-based region can be considered as a subgraph of the program dependence graph of method *m* which contains as dependence edges only the edges that start from and also end to nodes of the region. It should be noted that a loop-carried data dependence belongs to the region subgraph, if additionally the loop node through which the dependence is carried belongs to the nodes of the region. Formally, the edges belonging to region *R*(*B*) is set

$$E_B(R(B)) = \{p \rightarrow_c q \in E(m) \mid p, q \in R(B)\} \cup$$
$$\{p \rightarrow_d q \in E(m) \mid p, q \in R(B)\} \cup$$
$$\{p \rightarrow_{d(l)} q \in E(m) \mid l, p, q \in R(B)\}$$

where $E(m)$ is the set of all edges in the PDG of method *m*,
$p \rightarrow_c q$ denotes a control dependence edge from node *p* to node *q*,

$p \rightarrow_d q$ denotes a loop-independent data dependence edge from node p to node q, and $p \rightarrow_{d(l)} q$ denotes a loop-carried data dependence edge from node p to node q which is carried by loop l.

Assuming that slicing criterion (n, u) is given, which consists of statement n belonging to method m and variable u that is defined or used in statement n, it is necessary to determine the block-based regions in which a slice can be computed for the specific slicing criterion. To achieve this, the *control dependence graph* (i.e., the program dependence graph containing only control dependence edges) of method m is constructed. Figure 5.7 shows the control dependence graph of method `statement()` decomposed into basic blocks (block-based CDG). The control dependence graph actually represents the nesting of statements inside a method (assuming that the code does not include unstructured control flow or exception flow).

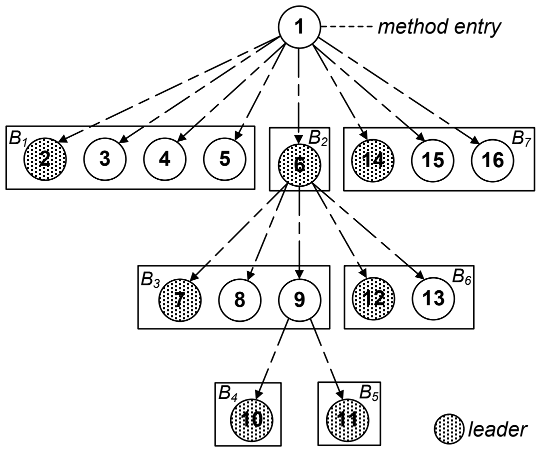


Figure 5.7: Control dependence graph of method `statement()`.

Assuming that node r is the node that directly dominates the leader node of basic block B, Maruyama defined as *dominated blocks* (*Dom*(B)) for basic block B, the set of blocks that are dominated by node r (a block is considered dominated by r if there exists a transitive control dependence from r to this block). For example, the leader node of block B_3 (node 7) is directly dominated by node 6 in the control dependence graph of Figure 5.7. As a result, the dominated blocks for basic block B_3 are the blocks that are dominated by node 6, namely $\{B_3, B_4, B_5, B_6\}$.

The sets of reachable and dominated blocks are used to compute the set of *boundary blocks* for statement n *Blocks*(n) in the following way:

1. For each basic block B of method m compute the sets of blocks *Reach*(B) and *Dom*(B).
2. If the basic block of statement n is contained in set $\{Reach(B) \cap Dom(B)\}$, then block B is added to the set of boundary blocks for statement n.

For example, the boundary blocks for statement 8 in Figure 5.6, which belongs to basic block B_3, is set $Blocks(8) = \{B_1, B_2, B_3\}$, since block B_3 is contained in the intersection of reachable and dominated blocks for basic blocks B_1, B_2 and B_3.

The block-based regions in which a slice can be computed for slicing criterion (n, u) are the regions of the boundary blocks for statement n (*Blocks*(n)). For example, the block-based regions for slicing criterion (8, `thisAmount`) are $R(B_1)$, $R(B_2)$ and $R(B_3)$, since the boundary blocks for statement 8 is set $Blocks(8) = \{B_1, B_2, B_3\}$.

5.3.3 Algorithms for the identification of Extract Method refactoring opportunities

The proposed approach provides two main algorithms for the identification of Extract Method refactoring opportunities. The first algorithm identifies refactoring opportunities where the complete computation of a local variable or parameter (*complete computation slice*) can be extracted, meaning that the resulting slice will contain all the assignment statements modifying the value of the local variable. The second algorithm identifies refactoring opportunities where all the statements affecting the state of an object (*object state slice*) can be extracted. The object reference can be a local variable which is declared inside the body of the original method, a parameter of the original method, or a field of the class containing the original method. Both algorithms do not require any user input (i.e., selection of statements or variables) in order to operate.

5.3.3.1 Identification of complete computation slices

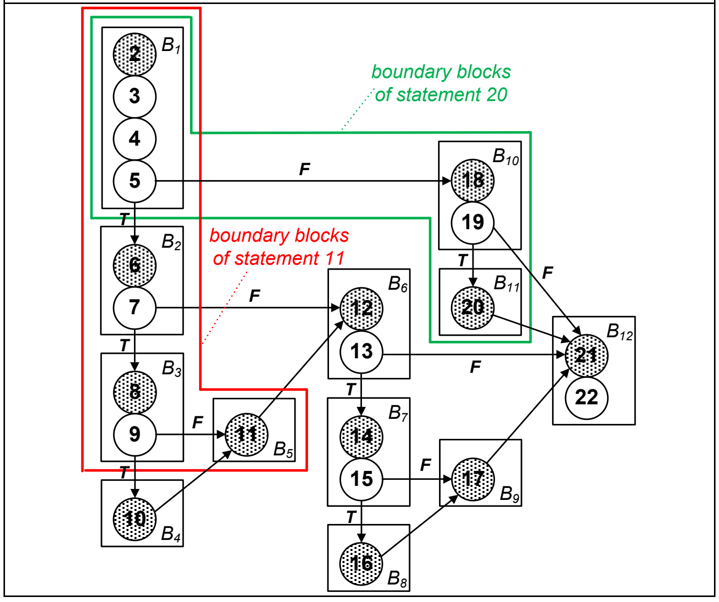
The proposed algorithm takes as input a method declaration m and returns a set of slice extraction refactoring suggestions for each variable declared inside method m whose value is modified by at least one assignment statement, covering the complete computation of the corresponding variable. The algorithm consists of the following steps:

1. Identify the set of local variables V which are declared inside method m.
2. For each variable $v \in V$ identify the set of seed statements C which contain an assignment of variable v. These statements along with variable v form a set of slicing criteria (c, v), where $c \in C$.
3. For each statement $c \in C$ compute the set of boundary blocks $Blocks(c)$.
4. Calculate the common boundary blocks for the statements in set C as $Blocks(C) = \bigcap_{c \in C} Blocks(c)$.
5. For each slicing criterion (c, v), where $c \in C$, and boundary block $B_n \in Blocks(C)$ compute the block-based slices $S_B(c, v, B_n)$. Block-based slice $S_B(c, v, B_n)$ is the set of statements that may affect the computation of variable v at statement c (backward slice), extracted from the program dependence subgraph corresponding to region $R(B_n)$.
6. For each $B_n \in Blocks(C)$ the union of slices $US_B(C, v, B_n) = \bigcup_{c \in C} S_B(c, v, B_n)$ is a slice that covers the complete computation of variable v within the region $R(B_n)$.

This algorithm produces for each variable v declared inside method m, a number of slices which is equal to the size of $Blocks(C)$, where C is the set of statements containing an assignment of variable v.

The application of the algorithm will be demonstrated on a well-established refactoring teaching example [28]. Figure 5.8 illustrates method `printDocument()` and its control flow graph decomposed into basic blocks.

```
1   public void printDocument(Packet document) {
2     String author = "Unknown";
3     String title = "Untitled";
4     int startPos = 0, endPos = 0;
5     if (document.message_.startsWith("!PS")) {
6       startPos = document.message_.indexOf("author:");
7       if (startPos >= 0) {
8         endPos = document.message_.indexOf(
          ".", startPos + 7);
9         if (endPos < 0)
10          endPos = document.message_.length();
11        author = document.message_.substring(
          startPos + 7, endPos);
        }
12      startPos = document.message_.indexOf("title:");
13      if (startPos >= 0) {
14        endPos = document.message_.indexOf(
          ".", startPos + 6);
15        if (endPos < 0)
16          endPos = document.message_.length();
17        title = document.message_.substring(
          startPos + 6, endPos);
        }
      } else {
18      title = "ASCII DOCUMENT";
19      if (document.message_.length() >= 16)
20        author = document.message_.substring(8, 16);
      }
21    System.out.println(author);
22    System.out.println(title);
    }
```



boundary blocks of statement 20
boundary blocks of statement 11
B1
B2
B3
B4
B5
B6
B7
B8
B9
B10
B11
B12
T
F

Figure 5.8: Method `printDocument()` and the corresponding control flow graph.

Assume that the computation of variable `author` is intended to be extracted as a separate method. The algorithm is applied as follows:

a. The assignment statements of variable `author` are statements 11 and 20 (underlined in the code of Figure 5.8).

b. The sets of boundary blocks for statements 11 and 20 are *Blocks*(11) = {B_1, B_2, B_3, B_5} and *Blocks*(20) = {B_1, B_{10}, B_{11}}, respectively (as shown in the control flow graph of Figure 5.8).

c. The intersection of the two sets of boundary blocks is *Blocks*({11, 20}) = {B_1} and as a result only block-based region $R(B_1)$ can be used as region for the union of the resulting static slices.
d. The block-based static slices for statements 11 and 20 are S_B(11, `author`, B_1) = {2, 4, 5, 6, 7, 8, 9, 10, 11} and S_B(20, `author`, B_1) = {2, 5, 19, 20}, respectively.
e. The union of the static slices is US_B({11, 20}, `author`, B_1) = {2, 4, 5, 6, 7, 8, 9, 10, 11, 19, 20}.

5.3.3.2 Identification of object state slices

The proposed algorithm takes as input a method *m* and returns a set of slice extraction refactoring suggestions for each reference inside method *m* pointing to an object whose state is affected by at least one statement containing an appropriate method invocation. The algorithm consists of the following steps:

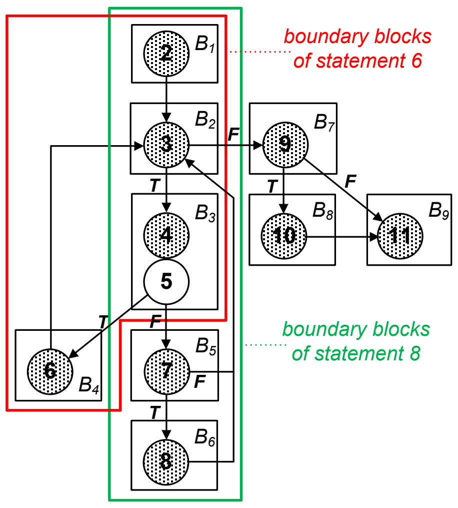
1. Identify the set of object references *R* existing inside method *m*. These references are local variables, parameters of *m*, or fields of the class containing *m* having a non-primitive type.
2. For each object reference $r \in R$ identify the set of fields F_r which are modified through reference *r* by method invocations inside the body of *m*. This is achieved by searching in the defined variables of each statement for composite variables having reference *r* as first part.
3. For each field $f \in F_r$ identify the set of seed statements C_f within the body of *m* that contain *f* in their set of defined variables. These statements along with variable *f* form a set of slicing criteria (c, f), where $c \in C_f$.
4. For each statement $c \in C_f$ compute the set of boundary blocks *Blocks*(*c*).
5. Calculate the common boundary blocks for the statements in each set C_f (referring to defined variable *f*) as $Blocks(C_f) = \bigcap_{c \in C_f} Blocks(c)$.
6. Calculate the common boundary blocks for all $Blocks(C_f)$, $\forall f \in F_r$ (referring to object reference *r*) as $Blocks(r) = \bigcap_{f \in F_r} Blocks(C_f)$.
7. For each slicing criterion (c, f), where $c \in C_f$, $f \in F_r$ and boundary block $B_n \in Blocks(r)$ compute the block-based slices $S_B(c, f, B_n)$. Block-based slice $S_B(c, f, B_n)$ is the backward slice extracted from the program dependence subgraph corresponding to region $R(B_n)$.
8. For each $B_n \in Blocks(r)$ the union of slices for field *f* is $US_B(C_f, f, B_n) = \bigcup_{c \in C_f} S_B(c, f, B_n)$.
9. For each $B_n \in Blocks(r)$ the union of slices for reference *r* $US_B(r, B_n) = \bigcup_{f \in F_r} US_B(C_f, f, B_n)$ is a slice that contains all the statements in method *m* affecting the state of the object referenced by *r*.

This algorithm produces for each reference *r*, a number of slices which is equal to the size of *Blocks*(*r*).

The application of the algorithm will be demonstrated on a real example taken from an open-source project, namely Violet 0.16 [47]. Figure 5.9 illustrates method `removeSelected()` and its control flow graph decomposed into basic blocks.

```
public class GraphPanel extends JPanel {
  private Graph graph;
  private Set selectedItems;

1 public void removeSelected() {
2   Iterator iter = selectedItems.iterator();
3   while (iter.hasNext()) {
4     Object selected = iter.next();
5     if (selected instanceof Node) {
6       graph.removeNode((Node) selected);
      }
7     else if (selected instanceof Edge) {
8       graph.removeEdge((Edge) selected);
      }
    }
9   if (selectedItems.size() > 0)
10    setModified(true);
11  repaint();
  }
}
```



boundary blocks of statement 6
boundary blocks of statement 8
B_1
B_2
B_3
B_4
B_5
B_6
B_7
B_8
B_9
T
F

```
public abstract class Graph {
  private ArrayList nodes;
  private ArrayList edges;
  private transient ArrayList nodesToBeRemoved;
  private transient ArrayList edgesToBeRemoved;
  private transient boolean needsLayout;

  public void removeNode(Node n) {
    if (nodesToBeRemoved.contains(n)) return;
    nodesToBeRemoved.add(n);
    for (int i = 0; i < nodes.size(); i++) {
      Node n2 = (Node) nodes.get(i);
      n2.removeNode(this, n);
    }
    for (int i = 0; i < edges.size(); i++) {
      Edge e = (Edge)edges.get(i);
      if (e.getStart() == n || e.getEnd() == n)
        removeEdge(e);
    }
    needsLayout = true;
  }
  public void removeEdge(Edge e) {
    if (edgesToBeRemoved.contains(e)) return;
    edgesToBeRemoved.add(e);
    for (int i = nodes.size() - 1; i >= 0; i--) {
      Node n = (Node)nodes.get(i);
      n.removeEdge(this, e);
    }
    needsLayout = true;
  }
}
```

Figure 5.9: Method `removeSelected()` and the corresponding control flow graph.

Assume that the statements affecting the state of the object referenced by field `graph` are intended to be extracted as a separate method. The algorithm is applied as follows:

a. The set of fields F_{graph} which are modified through reference *graph* consists of the following composite variables:
   1. *graph.nodesToBeRemoved.elementData*
   2. *graph.nodesToBeRemoved.size*
   3. *graph.nodesToBeRemoved.modCount*
   4. *graph.needsLayout*
   5. *graph.edgesToBeRemoved.elementData*
   6. *graph.edgesToBeRemoved.size*

    7. *graph.edgesToBeRemoved.modCount*

b. Fields 1-3 are defined at statement 6, while fields 4-7 are defined at statements 6 and 8. The resulting slicing criteria are eleven in total, based on the following sets of seed statements:
    1. $C_{graph.nodesToBeRemoved.elementData} = \{6\}$
    2. $C_{graph.nodesToBeRemoved.size} = \{6\}$
    3. $C_{graph.nodesToBeRemoved.modCount} = \{6\}$
    4. $C_{graph.needsLayout} = \{6, 8\}$
    5. $C_{graph.edgesToBeRemoved.elementData} = \{6, 8\}$
    6. $C_{graph.edgesToBeRemoved.size} = \{6, 8\}$
    7. $C_{graph.edgesToBeRemoved.modCount} = \{6, 8\}$
c. The sets of boundary blocks for statements 6 and 8 are $Blocks(6) = \{B_1, B_2, B_3, B_4\}$ and $Blocks(8) = \{B_1, B_2, B_3, B_5, B_6\}$, respectively (as shown in the control flow graph of Figure 5.9).
d. The resulting intersections of basic blocks are:
    1. $Blocks(C_{graph.nodesToBeRemoved.elementData}) = \{B_1, B_2, B_3, B_4\}$
    2. $Blocks(C_{graph.nodesToBeRemoved.size}) = \{B_1, B_2, B_3, B_4\}$
    3. $Blocks(C_{graph.nodesToBeRemoved.modCount}) = \{B_1, B_2, B_3, B_4\}$
    4. $Blocks(C_{graph.needsLayout}) = \{B_1, B_2, B_3\}$
    5. $Blocks(C_{graph.edgesToBeRemoved.elementData}) = \{B_1, B_2, B_3\}$
    6. $Blocks(C_{graph.edgesToBeRemoved.size}) = \{B_1, B_2, B_3\}$
    7. $Blocks(C_{graph.edgesToBeRemoved.modCount}) = \{B_1, B_2, B_3\}$
e. The final intersection of basic blocks is $Blocks(graph) = \{B_1, B_2, B_3\}$ and as a result block-based regions $R(B_1)$, $R(B_2)$ and $R(B_3)$ can be used as regions for the union of the resulting static slices.
f. In this code example, the resulting slices are the same for all slicing criteria. More specifically, $S_B(c, f, B_1) = \{2, 3, 4, 5, 6, 7, 8\}$, $S_B(c, f, B_2) = \{3, 4, 5, 6, 7, 8\}$ and $S_B(c, f, B_3) = \{4, 5, 6, 7, 8\}$, where $f \in F_{graph}$ and $c \in C_f$.
g. Consequently, the resulting unions of slices are also the same for all fields belonging to F_{graph}. More specifically, $US_B(C_f, f, B_1) = \{2, 3, 4, 5, 6, 7, 8\}$, $US_B(C_f, f, B_2) = \{3, 4, 5, 6, 7, 8\}$ and $US_B(C_f, f, B_3) = \{4, 5, 6, 7, 8\}$, where $f \in F_{graph}$.
h. Finally, the unions of slices for reference *graph* are $US_B(graph, B_1) = \{2, 3, 4, 5, 6, 7, 8\}$, $US_B(graph, B_2) = \{3, 4, 5, 6, 7, 8\}$ and $US_B(graph, B_3) = \{4, 5, 6, 7, 8\}$, respectively.

As it can be observed from the code of method `removeSelected()` in Figure 5.9, statements 2-8 exhibit a distinct functionality compared to the rest of the statements, which is related with the removal of the selected nodes and edges from the graph object corresponding to field `graph`.

5.3.3.3 Determination of indispensable statements and parameters of the extracted method

Indispensable statements are statements that belong to a given slice but should not be removed from the original method after slice extraction to assure that the original method remains operational (i.e., are statements required by the statements that remain in the original method in order to operate correctly). Indispensable statements are computed in the following way based on Maruyama's approach [70]:

Let $N(m)$ be the set of all nodes and $E(m)$ the set of all edges in the PDG of method m. Let S_B be a block-based slice resulting from the region of boundary block B

($R(B)$). Let U_B be the set of remaining nodes after the nodes of S_B are removed from $N(m)$ ($U_B = \{N(m) \setminus S_B\}$).

Let N_{CD} be the set of nodes belonging to S_B on which nodes belonging to U_B are control dependent (i.e., there exists a control dependence edge from a node in S_B to a node in U_B).

$N_{CD}(S_B, U_B) = \{p \in N(m) \mid p \rightarrow_c q \in E(m) \wedge p \in S_B \wedge q \in U_B\}$, where $p \rightarrow_c q$ denotes a control dependence from node p to node q.

Let N_{DD} be the set of nodes belonging to S_B on which nodes belonging to U_B are data dependent (i.e., there exists a data dependence edge from a node in S_B to a node in U_B) due to a variable other than the variable of the slicing criterion.

$N_{DD}(S_B, U_B, v) = \{p \in N(m) \mid p \rightarrow_d^u q \in E(m) \wedge u \neq v \wedge p \in S_B \wedge q \in U_B\}$, where $p \rightarrow_d^u q$ denotes a data dependence from node p to node q due to variable u and v is the variable of the slicing criterion.

Control indispensable nodes I_{CD} are the nodes of the slices that result using (p, u, B) as slicing criteria, where $p \in N_{CD}(S_B, U_B)$ and u belongs to the used variables of node p.

$I_{CD}(S_B, U_B) = \{q \in N(m) \mid q \in S_B(p, u, B) \wedge p \in N_{CD}(S_B, U_B) \wedge u \in Use(p)\}$

Data indispensable nodes I_{DD} are the nodes of the slices that result using (p, u, B) as slicing criteria, where $p \in N_{DD}(S_B, U_B, v)$ and u belongs to the defined variables of node p.

$I_{DD}(S_B, U_B, v) = \{q \in N(m) \mid q \in S_B(p, u, B) \wedge p \in N_{DD}(S_B, U_B, v) \wedge u \in Def(p)\}$

Eventually, the indispensable nodes I_B is the set resulting from the union of I_{CD} and I_{DD} sets ($I_B = I_{CD} \cup I_{DD}$). Indispensable nodes will be duplicated in both the original and the extracted method after slice extraction, while the set of nodes that can be actually removed from the original method is $\{S_B \setminus I_B\}$ and the set of nodes that actually remain in the original method is $\{U_B \cup I_B\}$.

The parameters of the extracted method are the variables of the original method for which a data dependence exists from the set of remaining nodes U_B to the set of slice nodes S_B. Formally, $P(S_B, U_B) = \{u \in V(m) \mid p \rightarrow_d^u q \in E(m) \wedge p \in U_B \wedge q \in S_B\}$, where $V(m)$ is the set of variables which are declared within the body of method m (including the parameters of method m).

5.3.4 Rules regarding behavior preservation and usefulness of the extracted functionality

The slices resulting from the algorithms of Section 5.3.3 are examined against a set of rules that exclude from being suggested as refactoring opportunities cases of slices whose extraction could possibly cause a change in program behavior. The rules are preventive in the sense that they prescribe conditions that should not hold in order to obtain extractable slices which preserve program behavior. Moreover, there is a category of rules which are used to reject some extreme cases of slices that lead to extracted methods with limited usefulness in terms of functionality.

5.3.4.1 Duplication of statements affecting the state of an object

In object-oriented code the invocation of a method can change the state of the object being referenced. This change in object state may in turn affect the execution of the code that follows in a method. Obviously, the duplication of such method invocations in both the remaining and the extracted method is not preserving the behavior of the program, since a duplicated statement is executed twice (i.e., once in the remaining method and once in the extracted method). To support this argument, two slice extraction examples based on the code of Figure 5.6 will be demonstrated. Both examples concern the extraction of code from method `statement()` using the same slicing criterion (10, `frequentRenterPoints`) but different block-based regions. The set of boundary blocks for statement 10 is $Blocks(10) = \{B_1, B_2, B_3, B_4\}$ (the layout of blocks is shown in Figure 5.6), and as a result, four block-based slices can be derived from this slicing criterion. Figure 5.10 shows the remaining and the extracted method when block-based slice S_B(10, `frequentRenterPoints`, B_2) is used.

```
1  public String statement() {
2    double totalAmount = 0;
3    int frequentRenterPoints = 0;
4    Enumeration rentals = _rentals.elements();
5    String result = "Rental Record for " + getName() + "\n";
     frequentRenterPoints = getFrequentRenterPoints(
       frequentRenterPoints, rentals);
6    while(rentals.hasMoreElements()) {
7      Rental each = (Rental) rentals.nextElement();
8      double thisAmount = each.getCharge();
12     result += "\t" + each.getMovie().getTitle() + "\t"
       + String.valueOf(thisAmount) + "\n";
13     totalAmount += thisAmount;
     }
14   result += "Amount owed is "
     + String.valueOf(totalAmount) + "\n";
15   result += "You earned " + String.valueOf(frequentRenterPoints)
     + " frequent renter points";
16   return result;
   }

   private int getFrequentRenterPoints(int frequentRenterPoints,
                Enumeration rentals) {
6    while(rentals.hasMoreElements()) {
7      Rental each = (Rental) rentals.nextElement();
9      if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE
            && each.getDaysRented() > 1)
10       frequentRenterPoints += 2;
       else
11       frequentRenterPoints++;
     }
     return frequentRenterPoints;
   }
```

Figure 5.10: Extraction of block-based slice S_B(10, `frequentRenterPoints`, B_2).

As it can be observed from Figure 5.10, after the execution of the extracted method `getFrequentRenterPoints()` the Enumeration `rentals` will not have any more elements to provide, since the `while` loop inside the extracted method has already iterated over all the elements of the enumeration. As a result, the `while` loop that follows inside method `statement()` will not be executed, since the invocation of method `hasMoreElements()` will return false. Obviously, in this case the behavior of the program is not preserved after slice extraction. The reason causing the change of behavior is that the invocation of method `nextElement()` in statement 7 affects the internal state of object reference `rentals` and at the same time statement

7 is duplicated in both the remaining and the extracted method. An alternative slice extraction using block-based slice S_B(10, `frequentRenterPoints`, B_1) is shown in Figure 5.11.

```
1  | public String statement() {
   |   int frequentRenterPoints = getFrequentRenterPoints();
2  |   double totalAmount = 0;
4  |   Enumeration rentals = _rentals.elements();
5  |   String result = "Rental Record for " + getName() + "\n";
6  |   while(rentals.hasMoreElements()) {
7  |     Rental each = (Rental) rentals.nextElement();
8  |     double thisAmount = each.getCharge();
12 |     result += "\t" + each.getMovie().getTitle() + "\t"
   |     + String.valueOf(thisAmount) + "\n";
13 |     totalAmount += thisAmount;
   |   }
14 |   result += "Amount owed is "
   |   + String.valueOf(totalAmount) + "\n";
15 |   result += "You earned " + String.valueOf(frequentRenterPoints)
   |   + " frequent renter points";
16 |   return result;
   | }
   |
   | private int getFrequentRenterPoints() {
3  |   int frequentRenterPoints = 0;
4  |   Enumeration rentals = _rentals.elements();
6  |   while(rentals.hasMoreElements()) {
7  |     Rental each = (Rental) rentals.nextElement();
9  |     if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE
   |            && each.getDaysRented() > 1)
10 |       frequentRenterPoints += 2;
   |     else
11 |       frequentRenterPoints++;
   |   }
   |   return frequentRenterPoints;
   | }
```

Figure 5.11: Extraction of block-based slice S_B(10, `frequentRenterPoints`, B_1).

As it can be observed from Figure 5.11, the slice extraction based on basic block B_1, where slicing covers the whole source method, preserves the behavior of the program in contrast with the slice extraction based on basic block B_2. The reason causing the preservation of behavior is that apart from statement 7, the declaration of object reference `rentals` (statement 4) is also duplicated in both the remaining and the extracted method. As a result, the `while` loops in the remaining and the extracted method iterate over two different `Enumeration` references derived from the same `Vector` object (field `_rentals`).

Rule 1: The duplicated statements (i.e., the statements corresponding to the set of indispensable nodes I_B) should not contain composite field variables (i.e., composite variables whose first part is an object reference existing in the original method and last part is a field modified through this specific object reference) in their sets of defined variables. From this rule are excluded the local object references whose declaration statement is also included to the duplicated statements.

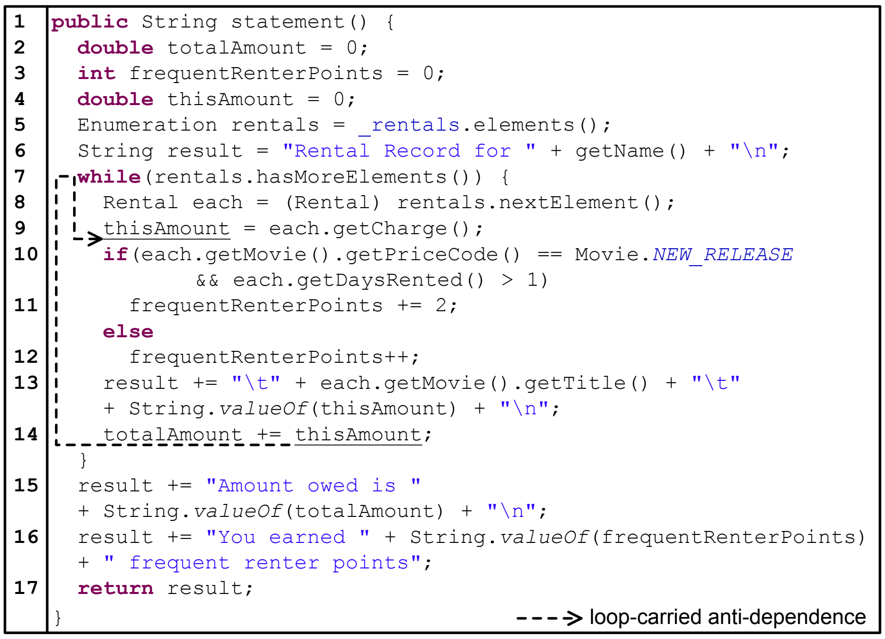
5.3.4.2 Duplication of statements containing a class instance creation

In the same manner that a statement causing a change in the state of an object can be duplicated, a statement creating an object may also be duplicated. Let us assume that a statement initializing or assigning reference *r* with a class instantiation (i.e., *r* = *new Type()*) is duplicated in both the original and the extracted method. Then each reference *r* (one being in scope within the original method and the other within the extracted method) will be referring to a different object in memory. As a result, the existence of non-duplicated statements affecting the state of the reference existing in the original method or the extracted method would cause an inconsistency of state between the two references. Such an inconsistency could in turn affect statements depending on reference *r*, causing a change in the behavior of the program.

Rule 2: A duplicated statement (i.e., a statement belonging to the set of indispensable nodes I_B) initializing or assigning object reference *r* with a class instantiation, should not have a data dependence due to variable *r* that ends to a statement of the removable nodes $\{S_B \setminus I_B\}$.

5.3.4.3 Preservation of existing anti-dependences

Another case that may cause change in behavior is the existence of an anti-dependence between a statement that remains in the original method and a statement belonging to the slice statements that will be removed from the original method. An anti-dependence exists from statement *p* to statement *q* (or statement *q* anti-depends on *p*) due to variable *x*, when there is a control flow path starting from statement *p* that uses the value of *x* and ending to statement *q* that modifies the value of *x* (regardless of any intermediate statements that may use the value of variable *x*). Just like data flow dependences, anti-dependences can be either loop carried (i.e., carried by a specific loop) or loop independent. Figure 5.12 shows an example of code containing a loop carried anti-dependence which is carried by the `while` loop in statement 7.



```
1   public String statement() {
2     double totalAmount = 0;
3     int frequentRenterPoints = 0;
4     double thisAmount = 0;
5     Enumeration rentals = _rentals.elements();
6     String result = "Rental Record for " + getName() + "\n";
7     while(rentals.hasMoreElements()) {
8       Rental each = (Rental) rentals.nextElement();
9       thisAmount = each.getCharge();
10      if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE
             && each.getDaysRented() > 1)
11        frequentRenterPoints += 2;
        else
12        frequentRenterPoints++;
13      result += "\t" + each.getMovie().getTitle() + "\t"
        + String.valueOf(thisAmount) + "\n";
14      totalAmount += thisAmount;
      }
15    result += "Amount owed is "
      + String.valueOf(totalAmount) + "\n";
16    result += "You earned " + String.valueOf(frequentRenterPoints)
      + " frequent renter points";
17    return result;
    }
```

Figure 5.12: Code example containing a loop carried anti-dependence.

As it can be observed, the value of variable `thisAmount` is used at statement 14 and in the next iteration of the `while` loop its value is modified at statement 9. Let us consider that slicing criterion (9, `thisAmount`) is used for the code of Figure 5.12. The set of boundary blocks for statement 9 is *Blocks*(9) = {B_1, B_2, B_3} (the layout of blocks is shown in Figure 5.6), and as a result, three block-based slices can be derived from this slicing criterion. The slice corresponding to the block-based region of block B_1 (region $R(B_1)$ contains all the statements of the original method) is S_B(9, `thisAmount`, B_1) = {4, 5, 7, 8, 9} and is extracted as shown in Figure 5.13.

```
1   public String statement() {
      double thisAmount = getThisAmount();
2     double totalAmount = 0;
3     int frequentRenterPoints = 0;
5     Enumeration rentals = _rentals.elements();
6     String result = "Rental Record for " + getName() + "\n";
7     while(rentals.hasMoreElements()) {
8       Rental each = (Rental) rentals.nextElement();
10      if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE
               && each.getDaysRented() > 1)
11        frequentRenterPoints += 2;
        else
12        frequentRenterPoints++;
13      result += "\t" + each.getMovie().getTitle() + "\t"
        + String.valueOf(thisAmount) + "\n";
14      totalAmount += thisAmount;
      }
15    result += "Amount owed is "
      + String.valueOf(totalAmount) + "\n";
16    result += "You earned " + String.valueOf(frequentRenterPoints)
      + " frequent renter points";
17    return result;
    }

    private double getThisAmount() {
4     double thisAmount = 0;
5     Enumeration rentals = _rentals.elements();
7     while (rentals.hasMoreElements()) {
8       Rental each = (Rental) rentals.nextElement();
9       thisAmount = each.getCharge();
      }
      return thisAmount;
    }
```

Figure 5.13: Extraction of slice S_B(9, `thisAmount`, B_1) causing change in behavior.

As it can be observed from Figure 5.13, the behavior of the program is not preserved after the extraction of block-based slice S_B(9, `thisAmount`, B_1), since the extracted method returns the amount of charge corresponding to the last element of Vector `_rentals`. As a result, the value of variable `thisAmount`, which is used in statements 13 and 14 in the original method, is correct only in the last iteration of the `while` loop inside the original method. Obviously, the final values of variables `result` and `totalAmount` are affected due to the incorrect value of variable `thisAmount` at each iteration. The reason causing this change in behavior is that the anti-dependence that initially existed in the original method is lost after slice extraction, since the statement from which it started remains in the original method while the statement to which it ended is moved to the extracted method.

Rule 3: There should not exist an anti-dependence within the block-based region $R(B)$ of slice S_B starting from a statement of the remaining nodes $\{U_B \cup I_B\}$ and ending to a statement of the removable nodes $\{S_B \setminus I_B\}$. Formally, the rule is expressed as:

$\{p \rightarrow q \in A_B(R(B)) \mid p \in \{U_B \cup I_B\} \wedge q \in \{S_B \setminus I_B\}\} = \varnothing$ and
$A_B(R(B)) = \{p \rightarrow_a q \mid p, q \in R(B)\} \cup \{p \rightarrow_{a(l)} q \mid l, p, q \in R(B)\}$, where

$p \rightarrow_a q$ denotes a loop-independent anti-dependence edge from node p to node q, and $p \rightarrow_{a(l)} q$ denotes a loop-carried anti-dependence edge from node p to node q which is carried by loop l.

5.3.4.4 Rules regarding the usefulness of the extracted code in terms of functionality

The goal of the rules defined in this section is to prevent some extreme cases of slices from being suggested as refactoring opportunities. These rules are related with the extent of the slice compared to the number of seed statements and the size of the original method, the degree of code duplication and the variable which is returned by the original method.

a. The number of statements in the union of slice statements US_B should be greater than the number of seed statements used in slicing criteria. In the case where the number of statements in US_B is equal to the number of seed statements used in slicing criteria (this is actually the minimum number of statements that can be extracted), the extracted code would be algorithmically trivial, since no additional statements are required for the computation of a given variable (or by the statements affecting the state of a given object). This means that a slice should consist of two statements at minimum, assuming that a single seed statement is used.
b. The number of statements in the union of slice statements US_B should not be equal to the number of statements in the original method. In such a case the extracted method would be exactly the same as the original method.
c. The statements which are duplicated in both the original and the extracted method should not contain all the seed statements used in slicing criteria. If all the seed statements used in slicing criteria were duplicated, then the computation of a given variable (or the statements affecting the state of a given object) would exist in both the original and the extracted method making the extraction redundant.
d. The variable which is returned by the original method should be excluded from slice extraction. If the computation of a given variable (or the statements affecting the state of a given object) that is returned by the original method was extracted, then the extracted method would essentially have the functionality and purpose of the original method.

5.3.5 Handling of try/catch blocks, branching and throw statements

Try/catch blocks are used in Java as a means to handle exceptions caused at runtime. The `try` block contains code that could throw an exception, while the `catch` clauses contain code that is directly executed when an exception is thrown in the body of the `try` block. Each `catch` clause is responsible for a specific exception type. The proposed approach considers `try` blocks as ordinary blocks of code in the construction of the control flow and program dependence graphs. This means that the `try` block itself is not represented as a node in the graphs, but the statements included in its body are control dependent on the parent of the `try` block in the abstract syntax tree. The statements contained in the `catch` clauses are also not included in the graphs. In the process of slice extraction the statements of the slice are examined whether they have a `try` block as immediate parent in the abstract syntax tree. If at least one of the slice statements contains a method invocation that could throw an exception or directly throws an exception, then the parent `try` block (along with the corresponding `catch` clauses) is copied to the extracted method and all the slice statements having this specific `try` block as immediate parent in the abstract syntax tree are placed inside the body of the `try` block. A `try` block is removed from the original method if all the statements inside its body are moved to the extracted method or the statements that remain inside its body do not throw any exceptions (in this case the remaining statements are moved to the parent of the `try` block in the abstract syntax tree).

Unstructured control flow is achieved in Java by three kinds of branching statements. The `break` statement terminates the innermost loop and transfers the control flow to the statement following the innermost loop. The `continue` statement skips the current iteration of the innermost loop and transfers the control flow to the evaluation expression that controls the innermost loop. Finally, the `return` statement exits from the current method and transfers the control flow to the point where the method was invoked. In general, the problem caused by branching statements is that they cannot be included in slices, thus affecting slice precision. The reason behind the non-inclusion of branching statements in slices is that branching statements do not form control or data dependences with other statements in the program dependence graph of a method. Kumar and Horwitz [59] proposed the *augmented* program dependence graph (APDG) as a means to handle properly the branching statements. However, in order to construct the APDG is required to represent the branching statements as pseudo-predicates in the control flow graph. A pseudo-predicate node has two outgoing edges where the one (labeled as true) goes to the target of the jump and the other one (labeled as false) goes to the statement that would follow the branching statement if no branching occurred. Obviously, the handling of branching statements as pseudo-predicates affects the way that block-based regions are formed in the proposed approach, since block-partitioning depends on branching nodes (i.e., nodes having two or more outgoing flow edges) as explained in Section 5.3.2. Consequently, it is not possible to adopt the solution of APDG in this approach. Alternatively, the proposed approach applies a post-processing procedure after the original slice is obtained. For each `break` and `continue` statement, it examines whether the innermost loop is included in the slice. In such a case, the statements belonging to the slice of the branching statement (and expanding within the block-based region of the original slice) are included in the original slice. Furthermore, if the innermost loop contains statements inside its body that will eventually remain in the original method (i.e., statements belonging to set $\{U_B \cup I_B\}$), then the statements belonging to the slice of the branching statement are also added to indispensable statements (i.e., set I_B). This process cannot be applied to `return` statements, since the operation of a `return` statement is directly associated with the method that it belongs to, and thus a `return` statement cannot be copied to another method. As a result, if a `return` statement has a direct or indirect incoming control dependence from a statement belonging to a given slice, then this slice is rejected from being suggested as a refactoring opportunity.

Throw statements are special statements which are used for creating and throwing exception objects. Exception types are divided into *checked* exceptions which must be explicitly handled by a `catch` block or propagated up the call stack of methods (java.lang.Exception subclasses), and *unchecked* exceptions which do not have this requirement (java.lang.RuntimeException subclasses). Similarly to branching statements, `throw` statements do not form control or data dependences with other statements in the program dependence graph of a method and thus should be also handled in a special manner. The proposed approach applies a post-processing procedure after the original slice is obtained. If a `throw` statement is control dependent on a statement of the original slice, then the statements belonging to the slice of the `throw` statement (and expanding within the block-based region of the original slice) are included in the original slice. Furthermore, if a `throw` statement is control dependent on a statement that will eventually remain in the original method (i.e., statements belonging to set $\{U_B \cup I_B\}$), then the statements belonging to the slice of the `throw` statement are also added to indispensable statements (i.e., set I_B).

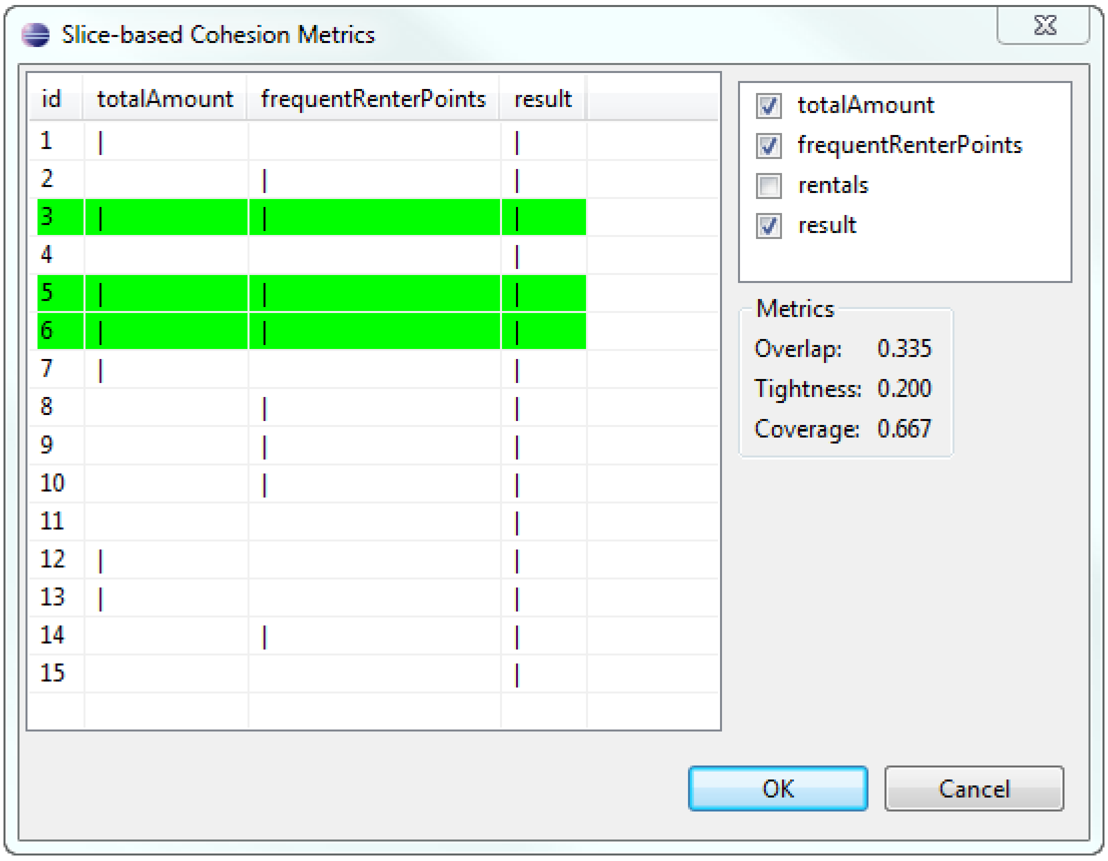
5.4 JDeodorant Eclipse plug-in

The proposed method has been implemented as an Eclipse plug-in [49] that identifies Extract Method refactoring opportunities on Java projects, highlights the code fragments suggested to be extracted (by indicating with green color the statements that will be moved to the extracted method and with red color the statements that will be duplicated in both the original and the extracted method) and automatically applies on source code the refactorings which are eventually approved by the user. In order to control the number and the quality of the identified refactoring opportunities being reported, JDeodorant offers a preference page where the user can define various threshold values regarding the following properties:

- The minimum size (in number of statements) that a method should consist of in order to be examined for potential refactoring opportunities.
- The minimum number of statements that a slice should consist of in order to be reported as a refactoring opportunity.
- The maximum number of duplicated statements (between the original and the extracted method) that the extraction of a slice may introduce in order to be reported as a refactoring opportunity.
- The maximum ratio of duplicated to extracted statements (ranging over the interval [0, 1]) that should apply for a slice extraction refactoring in order to be reported.

In order to support the user in assessing the cohesion of a given method (i.e., the degree of interdependence among the statements required for the computation of the variables declared inside a method), JDeodorant offers a flexible calculator for slice-based cohesion metrics [115]. The calculator automatically computes the backward slices for all the local variables whose scope is the block corresponding to the method body, constructs the slice profile [114] of the examined method and highlights the statements which are common to all computed slices, as shown in Figure 5.14. The user has the ability to exclude from the slice profile of the examined method any variables which cannot be considered as output variables (i.e., variables playing an auxiliary role in the computation of other variables and whose computation is not intended to be extracted in a separate method) in order to improve the accuracy of the calculated slice-based cohesion metrics. Figure 5.14 shows the slice profile and the calculated slice-based cohesion metrics, namely overlap, tightness and coverage for the method of Figure 5.6. As it can be observed, variable `rentals` has been excluded from the slice profile, since it plays an auxiliary role in the computation of the rest variables.

Finally, JDeodorant sorts the identified refactoring opportunities according to their effectiveness as measured by the *duplication ratio* (i.e., the ratio of the number of statements that will be duplicated after the extraction of a slice to the number of statements which are going to be extracted). First, the identified slice extraction opportunities are grouped according to the variable or object reference that they concern (i.e., a slice that can be extracted using more than one block-based regions is considered as a single refactoring opportunity) in order to present relevant refactoring opportunities in a consecutive way. The resulting groups are sorted according to the average duplication ratio of the refactoring opportunities belonging to each group in ascending order. In the case where two groups have an average duplication ratio equal to zero (i.e., none of the slice extraction opportunities belonging to the groups causes duplication of statements), the groups are sorted according to the maximum number of statements that can be extracted by the refactoring opportunities belonging to each group in descending order. The reasoning behind this sorting mechanism is that the extraction of slices causing significant duplication should be less preferred, since such slices are generally cohesive with the method from which they are extracted.



id	totalAmount	frequentRenterPoints	result
1	|		|
2		|	|
3	|	|	|
4			|
5	|	|	|
6	|	|	|
7	|		|
8		|	|
9		|	|
10		|	|
11			|
12	|		|
13	|		|
14		|	|
15			|

☑ totalAmount
☑ frequentRenterPoints
☐ rentals
☑ result

Metrics
Overlap: 0.335
Tightness: 0.200
Coverage: 0.667

Figure 5.14: JDeodorant calculator for slice-based cohesion metrics.

5.5 Evaluation

The evaluation of the proposed method consists of three parts, namely an independent assessment of the identified refactoring opportunities regarding their soundness and usefulness, an investigation of the impact of the suggested refactorings on slice-based cohesion metrics and finally an investigation of the impact of the suggested refactorings on the external behavior of the program.

The criteria for selecting an appropriate project for the evaluation of the proposed method are the following:

a. The source code of the project should be publicly available, since JDeodorant performs source code analysis in order to identify refactoring opportunities. Furthermore, source code availability will make possible the reproduction of the experimental results.
b. The project should be large enough in order to present a sufficient number of refactoring opportunities.

c. The project should exhibit high test coverage to make feasible the examination of behavior preservation after the application of the identified refactoring opportunities.

The project which has been selected is JFreechart. It is a rather mature open-source chart library which has been constantly evolving since 2002. Version 1.0.0 consists of 771 classes and 95K lines of source code (as measured by sloccount), while its average test coverage is 63.7% (as measured by EclEmma code coverage tool).

5.5.1 Independent assessment

To consider an approach identifying refactoring opportunities successful, it must be able to suggest refactorings which are conceptually sound and useful apart from preserving program behavior and having a positive impact on certain quality metrics. The conceptual soundness and usefulness of the refactoring opportunities can only be assessed by human expertise. To this end, an independent expert was asked to express his opinion on the refactoring opportunities that were identified in package org.jfree.chart of JFreeChart project. The independent designer had significant experience in software design (he has been working for more than 13 years as a telecommunications software designer) and deep knowledge of object-oriented design principles. More specifically, the independent designer had to answer the following questions for each identified refactoring opportunity:

a. Does the code fragment suggested to be extracted as a separate method have a distinct and independent functionality compared to the rest of the original method? If yes, describe its functionality by providing the name of the extracted method. If no, provide the reason for which the refactoring suggestion is not acceptable.
b. Does the application of the suggested refactoring solve an existing design flaw (e.g. by decomposing a complex method, removing a code fragment that is duplicated among several methods, or extracting a code fragment suffering from Feature Envy)?

Package org.jfree.chart (excluding its sub-packages) consists of 18 classes, 301 methods with body and 4564 lines of source code. It is actually the core package of JFreechart library, since it is responsible for generating all supported chart types. In order to obtain meaningful refactoring suggestions the methods having less than 10 statements and the refactoring opportunities corresponding to slices with less than 4 statements have been excluded from the report by activating the appropriate property thresholds. The activated threshold, which is related with the size of the methods being examined for the identification of refactoring opportunities, reduced the number of analyzed methods from 301 to 51, from which only 39 presented at least one refactoring opportunity. The results of the evaluation are summarized in Table 5.1.

Table 5.1: Independent assessment of the identified refactoring opportunities.

package	**number of refactoring opportunities**				
	identified	*having distinct functionality*	*removing duplicated code*	*decomposing a complex method*	*constituting a feature envy case*
org.jfree.chart	64	57	15	11	1

As it can be observed from Table 5.1, the independent designer supported that 57 out of 64 (89%) identified refactoring opportunities correspond to code fragments having a distinct functionality compared to the rest of the original method. The independent designer disapproved 7 out of 64 (11%) identified refactoring opportunities for the following reasons:

- The code fragment suggested to be extracted did not have an obvious functionality and thus the extracted method would not have a clear purpose. (2/7)
- The code fragment suggested to be extracted had a trivial functionality and thus the extracted method would be useless. (1/7)
- The code fragment suggested to be extracted covered a large portion of the original method and thus the remaining functionality in the original method would be very limited after its extraction. (2/7)
- The code fragment suggested to be extracted shared several statements with other slices in the original method and thus its extraction would cause significant code duplication between the remaining and the extracted method. (2/7)

Furthermore, the independent designer reported that 27 out of 64 (42%) identified refactoring opportunities actually resolved (or in some cases helped to resolve) an existing design flaw. More specifically, 15 refactoring opportunities were utilized to remove three groups of duplicated code. The largest group of duplicated code consists of 11 cases that were extracted into a single method. Finally, 11 refactoring opportunities were utilized to decompose complex methods and one refactoring opportunity resulted in an extracted method suffering from Feature Envy that should be further moved to the envied class.

5.5.2 Impact on slice-based cohesion metrics

The empirical study of Meyers and Binkley [72] has shown that slice-based metrics can be used to quantify the deterioration that accompanies software evolution and measure the progress of a reengineering effort. To provide an estimate of the improvement in terms of cohesion introduced by the decomposition of methods, the slice-based cohesion of the original method (before slice extraction), the remaining method (after slice extraction) and the extracted method have been measured for the refactoring opportunities that the independent designer has agreed on.

Ott and Thuss [115] were the first that formally defined a set of quantitative metrics in order to estimate the level of cohesion in a module. The defined cohesion metrics were based on *slice profiles* [114] which constitute a convenient representation for revealing slice patterns within a module. Let V_M be the set of variables used by module M and V_O be a subset of V_M containing only the output variables of M. As output variables are considered the variable which is returned by M, the global variables which are modified by M and the parameters which are passed by reference and are modified by M. Finally, let SL_i be the slice obtained for variable $v_i \in V_O$ and SL_{int} be the intersection of SL_i over all $v_i \in V_O$. The tightness, overlap and coverage of module M are defined as:

$$Tightness(M) = \frac{|SL_{\text{int}}|}{length(M)},\ Overlap(M) = \frac{1}{|V_O|}\sum_{i=1}^{|V_O|}\frac{|SL_{\text{int}}|}{|SL_i|},$$

$$Coverage(M) = \frac{1}{|V_O|}\sum_{i=1}^{|V_O|}\frac{|SL_i|}{length(M)}$$

Tightness expresses the ratio of the number of statements which are common to all slices over the module length, while overlap expresses the average ratio of the number of statements which are common to all slices to the size of each slice. The higher the tightness and overlap of a module is, the more cohesive the module is. Obviously, in modules with high tightness or overlap the number of duplicated statements between the remaining and the extracted method will be large after the extraction of a slice. On the other hand, coverage expresses the average slice size over the module length and thus is not directly associated with the degree of common statements among the slices. However, a high value of coverage, which can be achieved when the slices extend over a large portion of the module, indirectly indicates the existence of several common statements among the slices.

In the Java programming language only a single variable can be returned by a given method, since the parameters are passed by value and thus their initial value is not possible to change during the execution of the method. Obviously, using a single variable (i.e., the returned variable) in the slice profile of a method would result in artificially high values of slice-based cohesion metrics which would not sufficiently reveal the actual cohesion of the method. To overcome this problem, only the variables whose scope is the block corresponding to the body of the method under examination have been considered as output variables, since these variables could be potentially returned at the end of the method. Furthermore, the considered output variables which are simply accessed and not modified within the body of the method are excluded from the slice profile.

Table 5.2 shows the average improvement of slice-based cohesion metrics introduced from the application of the Extract Method refactorings which have been approved by the independent expert. More specifically, the values of the second column have been estimated by calculating the average difference of the corresponding metric between the remaining method (i.e., the original method after the application of the refactoring) and the original method. The third column indicates the average metric values for the extracted methods which have been created after the application of the refactorings. Finally, the values of the fourth column have been estimated by calculating the average difference of the corresponding metric between the average metric value for the extracted and the remaining method (i.e., the changed/created methods after the application of the refactoring) and the original method (i.e., the method existing before the application of the refactoring).

Table 5.2: Average improvement of slice-based cohesion metrics.

	remaining - original	**extracted**	**(extracted + remaining)/2 - original**
Overlap	+0.287	0.891	+0.303
Tightness	+0.190	0.827	+0.319
Coverage	-0.015	0.917	+0.113

As it can be observed from the second and fourth columns of Table 5.2, the improvement of slice-based cohesion metrics can be considered significant by taking into account that their values range over the [0, 1] interval. A slight deterioration is observed in the average difference of coverage between the remaining and the original method, which, however, is so minor, that coverage can be considered unchanged. Finally, as it can be observed from the third column of Table 5.2, the slice-based cohesion metrics for the extracted methods exhibit significantly high average values indicating that the corresponding complete computation and object state slices constitute strongly cohesive code fragments.

5.5.3 Impact on program behavior

To assess the impact of the identified refactoring opportunities on program behavior the corresponding refactoring transformations have been applied on source code and the JUnit tests of the project under examination have been run in order to find out whether the applied refactorings caused test errors. From the 39 methods presenting at least one refactoring opportunity in package org.jfree.chart of JFreeChart project, 21 of them were actually associated with unit tests having an average test code coverage equal to 87% (as measured by EclEmma code coverage tool). The average test code coverage percentage can be considered sufficiently high in order to assess the preservation of program behavior after the application of the refactorings.

In total, 41 refactoring opportunities were identified for the 21 methods being tested in package org.jfree.chart of JFreeChart project. After the application of each refactoring all unit tests of the project were executed in order to examine whether the applied refactoring caused test errors. All of the applied refactorings passed the tests successfully without causing any test failure. Therefore, it can be concluded with a relative certainty that the defined behavior preservation rules have successfully excluded refactoring opportunities that could possibly cause a change in program behavior.

Chapter 6

6 Employing Eclipse JDT Core and LTK in JDeodorant

This chapter concerns the insights gained from working with Eclipse Java Development Tools (JDT), while developing JDeodorant plug-in [49]. Hopefully, the gained insight might prove to be useful for researchers and developers willing to develop their own application using Eclipse JDT.

Eclipse JDT [33] contributes a set of plug-ins adding the capabilities of a full-featured Java IDE to the Eclipse platform. The JDT plug-ins are categorized into:

- *JDT APT* which adds annotation processing support to Java 5 projects in Eclipse.
- *JDT Core* which defines the non-UI infrastructure.
- *JDT Debug* which implements Java debugging support and works with any JDPA-compliant target Java VM.
- *JDT Text* which provides the Java editor.
- *JDT UI* which implements Java-specific workbench contributions, such as the Package Explorer, the Type Hierarchy View, the Java Outline View, and various Wizards for creating Java elements.

JDeodorant plug-in mainly employs APIs belonging to JDT Core. In general, the infrastructure offered by JDT Core includes:

- An incremental Java builder.
- A Java Model that provides API for navigating the Java element tree. The Java element tree defines a Java centric view of a project.
- Code assist and code select support.
- An indexed based search infrastructure that is used for searching, code assist, type hierarchy computation, and refactoring.

6.1 Representation of Java elements in JDT Core

Eclipse JDT uses two levels in order to represent Java elements. The first representation level is referred to as *Java Model* and can be considered as a high-level representation of a *Java Project*. The Java Model represents a Java Project in a tree structure. An example of a Java Model instance visualized in the Package Explorer view is shown in Figure 6.1.

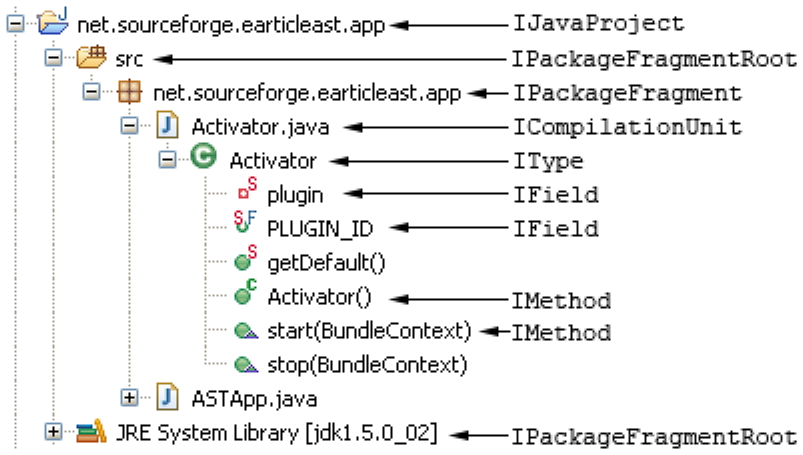


Figure 6.1: An example of a Java Model instance.

The nodes of the Java Model tree represent the following Java elements:

- *IJavaProject*: It is the root node of the Java Model and represents a Java Project. It contains *IPackageFragmentRoots* as child nodes.
- *IPackageFragmentRoot*: It represents a source or a class folder of a project, a .zip or a .jar file. An IPackageFragmentRoot can hold source or binary files.
- *IPackageFragment*: It represents a single package. It contains ICompilationUnits or IClassFiles, depending on the type of the IPackageFragmentRoot (i.e., source or binary, respectively). It should be noted that sub-packages are not placed as children of their parent package in the tree structure, but rather as children of the same IPackageFragmentRoot that their parent package belongs to. In other words an IPackageFragment does not have other IPackageFragments as children.
- *ICompilationUnit*: It represents a Java source file. An ICompilationUnit contains the set of top-level type declarations (IType) which are declared within its body.
- *IType*: It represents a type declaration. An IType contains the fields (IField), the methods (IMethod) and nested types (IType) which are declared within its body.
- *IMethod*: It represents the signature of a declared method (i.e., method name, return type, parameter types and names, and thrown exception types).
- *IField*: It represents a declared field (i.e., field type and name).
- *IClassFile*: It represents a Java bytecode file.

The Java Model is a lightweight structure, since it does not contain the bodies of methods in the tree structure (i.e., the highest detail level of the provided information is method signatures and fields). As a result, it requires a relatively small amount of memory and it can be re-created very fast in case of changes.

In order to obtain full access to Java source code down to statement level, JDT Core provides the Abstract Syntax Tree (AST) API [32]. The abstract syntax tree (in the form of a *CompilationUnit* AST node) for a given *ICompilationUnit* can be obtained as shown in the code of Figure 6.2.

```
void parseAST(ICompilationUnit unit) {
   ASTParser parser = ASTParser.newParser(AST.JLS3);
   parser.setKind(ASTParser.K_COMPILATION_UNIT);
   parser.setSource(unit);
   parser.setResolveBindings(true);
   CompilationUnit cu = (CompilationUnit)parser.createAST(null);
}
```

Figure 6.2: Abstract syntax tree creation for a given *ICompilationUnit*.

With statement `ASTParser.newParser(AST.JLS3)`, the parser is advised to parse the code following the Java Language Specification, Third Edition which includes the new syntax introduced in Java 5. The resulting *CompilationUnit* is the root node of the AST. All node types in the AST are subclasses of the *ASTNode* type and are grouped in inheritance hierarchies marked by the following abstract superclasses:

- *BodyDeclaration* superclass represents body declarations that may appear within the body of classes or interfaces. Some concrete subclasses of *BodyDeclaration* are the *TypeDeclaration*, *MethodDeclaration*, *FieldDeclaration*, *Initializer* (i.e., static initialization block), *EnumDeclaration* and *AnnotationTypeDeclaration*.
- *Type* superclass represents all kinds of types and has as concrete subclasses the *PrimitiveType* (i.e., byte, short, char, int, long, float, double, boolean and void types), *SimpleType*, *ArrayType*, *QualifiedType*, *ParameterizedType* (i.e., types with generics) and *WildcardType*.
- *Statement* superclass represents all kinds of statements that may exist within the body of a method. The statements can be divided to statements with body containing other statements such as the *Block*, *DoStatement*, *EnhancedForStatement*, *ForStatement*, *IfStatement*, *LabeledStatement*, *SwitchStatement*, *SynchronizedStatement*, *TryStatement* and *WhileStatement* and statements without body such as the *ExpressionStatement*, *ConstructorInvocation*, *SuperConstructorInvocation*, *EmptyStatement*, *ThrowStatement*, *VariableDeclarationStatement*, *ReturnStatement*, *AssertStatement*, *BreakStatement*, *ContinueStatement* and *SwitchCase*.
- *Expression* superclass represents all kinds of expressions that may exist within statements. Some subclasses of *Expression* are *Name*, *MethodInvocation*, *FieldAccess*, *ArrayCreation*, *ArrayAccess*, *Assignment*, *InstanceofExpression*, *ClassInstanceCreation*, *SuperMethodInvocation*, *SuperFieldAccess*, *InfixExpression*, *PostfixExpression*, *PrefixExpression*, *ThisExpression*, and *ConditionalExpression*.

The statement `parser.setResolveBindings(true)` requests that the compiler should provide binding information for the AST nodes it creates. Bindings are essential in program analysis, since they provide extended resolved information (derived from the compiler) for the named entities of a program. The most widely used bindings are the ones that refer to variables (*IVariableBinding* interface), types (*ITypeBinding* interface) and methods (*IMethodBinding* interface). The information that they provide is also given in the form of bindings. It is very important to mention that bindings are available only on code without syntax errors.

An *IVariableBinding* provides information about the type of the corresponding variable, whether the variable corresponds to a field, parameter, or local variable. Furthermore, if the variable corresponds to a parameter or local variable it provides information about the method containing the scope in which this variable is declared, and if the variable corresponds to a field it provides information about the class or interface that declares this field.

An *ITypeBinding* provides information about the fields, methods and types which are declared in the corresponding type, the superclass, implemented interfaces and modifiers of the corresponding type, the package in which the corresponding type is declared, etc.

An *IMethodBinding* provides information about the signature of the corresponding method (i.e., return type, parameter types and thrown exception types), the class or interface that declares the corresponding method, etc.

An example that illustrates the importance of bindings is the case where two variable names (i.e., two different *SimpleName* AST nodes) are examined whether they refer to the same variable. Obviously, the comparison of the AST nodes using method `equals()` of class *ASTNode* will return false, since method `equals()` returns true only if the AST node that invokes it and the AST node that is passed as argument actually refer to the same node (it should be noted that method `equals()` returns false when the compared AST nodes belong to abstract syntax trees resulting from different parsers, regardless of whether the compared nodes actually correspond to the same element in the code). The comparison of the names corresponding to the *SimpleName* AST nodes is not safe, since different variables having the same name may exist not only in different methods but also in the same method (within different scopes). The safest way to determine whether the two *SimpleName* AST nodes refer to the same variable is to examine whether their *IVariableBindings* are equal. The *IVariableBindings* can be resolved by invoking method `resolveBinding()` on each *SimpleName* AST node.

Another advantage of bindings is that they can be compared for equality (by using method `isEqualTo()`) even if the AST nodes from which they are resolved belong to different abstract syntax trees (e.g. ASTs resulting from different *ICompilationUnits*, or even from the same *ICompilationUnit* through different *ASTParsers*). An example that illustrates the importance of this feature is the case where a method invocation occurs to a compilation unit other than the one that the invoked method is declared. Obviously in this case, the *MethodInvocation* AST node and the corresponding *MethodDeclaration* AST node belong to different abstract syntax trees. The safest way to determine whether the *MethodInvocation* AST node corresponds to this specific *MethodDeclaration* AST node is to examine whether their *IMethodBindings* are equal using method `isEqualTo()` which compares the keys corresponding to the bindings. The binding key is a unique string representation of a binding that has the same value regardless of the abstract syntax tree from which the binding is resolved. The *IMethodBindings* can be resolved by invoking method `resolveBinding()` on each AST node.

However, the important help provided by bindings comes with the cost of expensive computation and significantly increased memory consumption. The increased memory requirements are caused by the fact that bindings are fully connected with each other [2]. This means that a binding holds references to a number of other bindings which in turn hold references to a number of other bindings and so on. As a result, a single binding may recursively hold references to hundreds of bindings. Consequently, AST nodes with binding information should not be kept permanently in memory.

6.2 Representation of Java elements in JDeodorant

JDeodorant is an application that heavily depends on AST. At a first level, it requires AST in order to perform program analysis on the source code of a Java project and identify potential refactoring opportunities. At a second level, it requires AST in order to apply the refactorings selected by the user on the source code of the examined Java project. As a result, AST information may be reused in several occasions for a different purpose. It becomes obvious that the architecture of JDeodorant should support the reuse of AST information without storing it permanently in memory.

This is achieved by providing an intermediate representation of the required Java elements and a mechanism that enables the recovery of AST nodes without storing them permanently in memory. Figure 6.3 shows the UML Class diagram of the intermediate representation involving high-level Java elements, such as the ones represented by the Java Model of JDT Core.

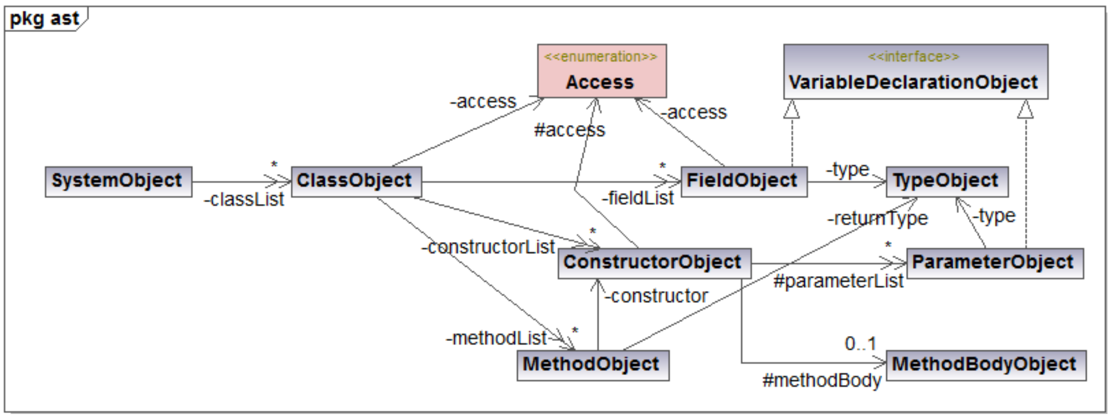


Figure 6.3: Representation of high-level Java elements.

Figure 6.4 shows the UML Class diagram of the intermediate representation involving Java elements that exist within the body of a method/constructor. As it can be observed from Figure 6.4, the statements within the body of a method are represented by means of the Composite design pattern [39]. Class *CompositeStatementObject* represents statements with body which may include other statements within their body, while class *StatementObject* represents leaf statements without body. Some composite statements, such as decision-making statements (if-then, if-then-else, switch) and loop statements (for, while, do-while) are associated with expressions which are used to control the program flow.

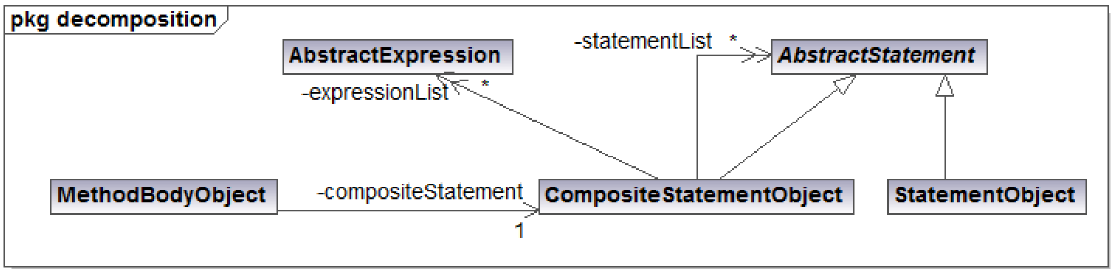


Figure 6.4: Representation of Java elements existing within the body of a method/constructor.

Figure 6.5 shows the UML Class diagram of the intermediate representation involving low-level Java elements, such as expressions required for program analysis. As it can be observed from Figure 6.5, a statement is associated with six kinds of expressions, namely method invocations, field accesses, super method invocations, local variable declarations, local variable accesses and creations of class instances and arrays. These kinds of expressions sufficiently support the program analysis requirements of the methodologies implemented in JDeodorant.

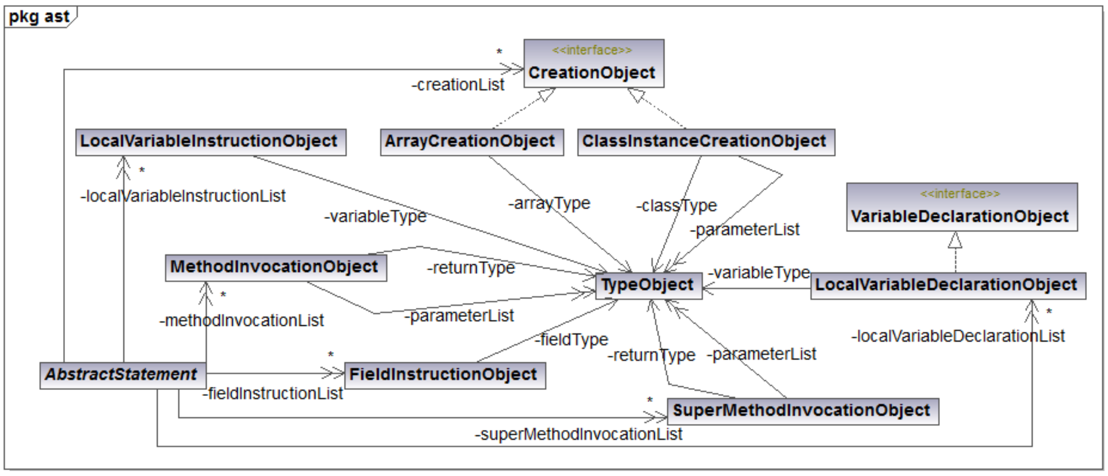


Figure 6.5: Representation of low-level Java elements.

Each class in the intermediate representation of Java elements corresponds to a specific type of AST node. Table 6.1 contains a mapping between the classes used for the intermediate representation of Java elements and the *ASTNodes* used in JDT Core.

Table 6.1: Mapping between classes in intermediate representation and AST nodes in JDT Core.

Java element in intermediate representation	AST node in JDT Core
ClassObject	TypeDeclaration
MethodObject, ConstructorObject	MethodDeclaration
FieldObject	VariableDeclarationFragment
ParameterObject	SingleVariableDeclaration
AbstractStatement	Statement
AbstractExpression	Expression
LocalVariableDeclarationObject	VariableDeclaration
MethodInvocationObject	MethodInvocation
SuperMethodInvocationObject	SuperMethodInvocation
FieldInstructionObject	SimpleName
LocalVariableInstructionName	SimpleName
ClassInstanceCreationObject	ClassInstanceCreation
ArrayCreationObject	ArrayCreation

The classes used for the intermediate representation of Java elements, shown in the first column of Table 6.1, do not hold a direct reference to the corresponding AST node, shown in the second column of Table 6.1. Instead, they are associated with class *ASTInformation*, shown in Figure 6.6, which holds lightweight information for a given AST node, such as the *ITypeRoot* (i.e., *ICompilationUnit* or *IClassFile*) that it belongs to, its start position in the abstract syntax tree and its length. The first time that an AST node is parsed and the corresponding object of the intermediate representation is created, an instance of *ASTInformation* is also created whose attributes are initialized with the aforementioned *ASTNode* properties (i.e., ITypeRoot, start position and length). The next time that this *ASTNode* is requested, the intermediate representation object invokes method `recoverASTNode()` through its reference to the already created instance of *ASTInformation*, retrieves the appropriate *ASTNode* object and casts it to the corresponding *ASTNode* type (as shown in Table 6.1). Method `recoverASTNode()` retrieves the *ASTNode* by invoking static method `NodeFinder.perform()` which takes three arguments, namely the *CompilationUnit* where the *ASTNode* belongs to, the start position of the *ASTNode* and the length of the *ASTNode*. The *CompilationUnit* can be regenerated through the *ASTParser* by setting as source the *ITypeRoot* attribute of the *ASTInformation* object.

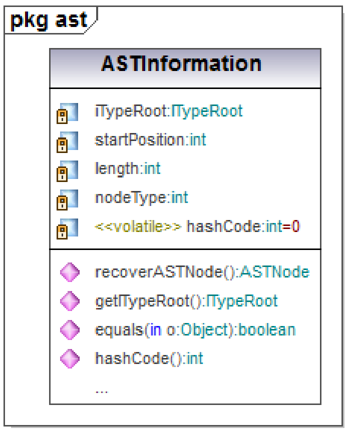


Figure 6.6: Class *ASTInformation* holding the properties required for the recovery of an *ASTNode*.

However, a continuous regeneration of *CompilationUnits* through *ASTParser* could raise significant performance issues. To face this problem, JDeodorant provides a *CompilationUnit* cache mechanism that takes into advantage the fact that when program analysis or refactoring is performed on a specific module, the requested AST nodes tend to belong to the same *CompilationUnit*. In order to avoid the consecutive regeneration of the same *CompilationUnit*, the cache mechanism holds a queue of *CompilationUnits* permanently in memory. If the requested AST node belongs to a *CompilationUnit* that is already inside the queue, then there is no need to regenerate the *CompilationUnit* through *ASTParser*. If the queue is full and the requested AST node does not belong to a *CompilationUnit* that is already present in the queue, then the oldest *CompilationUnit* is removed from the head of the queue and the new *CompilationUnit* is placed in the tail of the queue. The size of the queue depends on the specific requirements of the applied program analysis technique. JDeodorant uses a queue size ranging from 10 to 20 in order to support the requirements of the implemented methodologies. The benefits of the employed cache mechanism in memory consumption are very important, if we consider that a Java project may consist of hundreds or even thousands *CompilationUnits*.

6.3 Change notification for Java elements

In the case where the designer performs a suggested refactoring or makes manual changes on source code, the abstract syntax trees on which a change occurs undergo modifications in their structure. Since the Java elements of the intermediate representation are indirectly associated with AST nodes through their reference to *ASTInformation*, the elements belonging to *CompilationUnits* on which a change occurred should be updated in order to reflect the modified AST structure.

Eclipse JDT Core provides the *IElementChangedListener* interface as a means to receive notifications of changes to Java elements maintained by the Java Model (a listener can be registered to JDT Core through static method `JavaCore.addElementChangedListener()`). By implementing method `elementChanged(ElementChangedEvent event)` of this interface a client can be notified of changes to the structure or contents of the Java Model tree through the *ElementChangedEvent* parameter. Each *ElementChangedEvent* object is associated with an *IJavaElementDelta* object describing the changes that occurred to the involved elements. It should be emphasized that an *IJavaElementDelta* object should be treated in a recursive manner, since it represents the changes in a tree structure (similar to the Java Model structure) which is usually rooted at the *IJavaModel* level. A typical implementation of the *IElementChangedListener* interface that can be used in order to retrieve the *CompilationUnits* that have been changed, added or removed is shown in the code of Figure 6.7.

```
class ElementChangedListener implements IElementChangedListener {

   public void elementChanged(ElementChangedEvent event) {
      IJavaElementDelta javaElementDelta = event.getDelta();
      processDelta(javaElementDelta);
   }
   private void processDelta(IJavaElementDelta delta) {
      IJavaElement javaElement = delta.getElement();
      switch(javaElement.getElementType()) {
      case IJavaElement.JAVA_MODEL:
      case IJavaElement.JAVA_PROJECT:
      case IJavaElement.PACKAGE_FRAGMENT_ROOT:
      case IJavaElement.PACKAGE_FRAGMENT:
         IJavaElementDelta[] affectedChildren =
            delta.getAffectedChildren();
         for(IJavaElementDelta affectedChild : affectedChildren)
            processDelta(affectedChild);
         break;
      case IJavaElement.COMPILATION_UNIT:
         ICompilationUnit cu = (ICompilationUnit)javaElement;
         if(delta.getKind() == IJavaElementDelta.ADDED) {
            //compilationUnit has been added to the model
         }
         else if(delta.getKind() == IJavaElementDelta.REMOVED) {
            //compilationUnit has been removed from the model
         }
         else if(delta.getKind() == IJavaElementDelta.CHANGED) {
            if((delta.getFlags() &
                  IJavaElementDelta.F_FINE_GRAINED) != 0)
            //compilationUnit has been subject to structural changes
         }
      }
   }
}
```

Figure 6.7: Implementation of the *IElementChangedListener* interface that can be used to retrieve the *CompilationUnits* that have been changed, added or removed.

As it can be observed from Figure 6.7, method `processDelta()` is recursive. In the case where the *IJavaElementDelta* parameter corresponds to a Java element which is the *Java Model*, a *Java Project*, a *Package Fragment Root*, or a *Package Fragment*, method `processDelta()` is invoked for each affected child of the given Java element. In the case where the *IJavaElementDelta* parameter corresponds to a *CompilationUnit*, the delta kind (i.e., *ADDED*, *REMOVED* and *CHANGED*) indicates if the given *CompilationUnit* has been added, removed, or changed, respectively. It should be noted that a structural change to the members of a *CompilationUnit* exists only when the delta flag *F_FINE_GRAINED* is enabled.

Before each execution of a new refactoring opportunity identification procedure on a given Java project that has been already analyzed, the changed *CompilationUnits* are reparsed and the Java elements of the intermediate representation (i.e., *ClassObjects*) corresponding to the changed *CompilationUnits* are replaced with the newly generated ones. The added *CompilationUnits* are parsed for the first time and the resulting Java elements of the intermediate representation (i.e., *ClassObjects*) are added to the already existing ones. Finally, the Java elements of the intermediate representation (i.e., *ClassObjects*) corresponding to the removed *CompilationUnits* are removed from the already existing ones. In this way it is avoided to reparse a Java project that has been already analyzed in its entirety.

6.4 Collecting AST nodes of the same type

In many of the refactoring opportunity identification techniques implemented in JDeodorant, there was a need to collect all the AST nodes that exist within a given statement having a specific type of interest (e.g. the collection of all *IfStatements* within the body of a method). This problem requires a solution that takes advantage of a common algorithm that recursively traverses the AST structure starting from a given node and at the same time is able to collect the visited AST nodes having a specific type of interest. A part of the solution adopted by JDeodorant is shown in the code of Figure 6.8.

```java
public class StatementCollector {
   private StatementInstanceChecker instanceChecker;

   public List<Statement> getIfStatements(Statement statement) {
      instanceChecker = new InstanceOfIfStatement();
      return getStatements(statement);
   }

   private List<Statement> getStatements(Statement statement) {
      List<Statement> statementList = new ArrayList<Statement>();
      if(statement instanceof Block) {
         Block block = (Block)statement;
         List<Statement> blockStatements = block.statements();
         for(Statement blockStatement : blockStatements)
            statementList.addAll(getStatements(blockStatement));
         }
      else if(statement instanceof IfStatement) {
         IfStatement ifStatement = (IfStatement)statement;
         Statement thenStatement = ifStatement.getThenStatement();
         statementList.addAll(getStatements(thenStatement));
         if(ifStatement.getElseStatement() != null) {
            Statement elseStatement = ifStatement.getElseStatement();
            statementList.addAll(getStatements(elseStatement));
         }
         if(instanceChecker.instanceOf(ifStatement))
            statementList.add(ifStatement);
      }
      else if(statement instanceof ForStatement) {
         ForStatement forStatement = (ForStatement)statement;
         statementList.addAll(getStatements(forStatement.getBody()));
         if(instanceChecker.instanceOf(forStatement))
            statementList.add(forStatement);
      }
      else if(statement instanceof WhileStatement) {
         WhileStatement whileStatement = (WhileStatement)statement;
         Block whileBody = whileStatement.getBody();
         statementList.addAll(getStatements(whileBody));
         if(instanceChecker.instanceOf(whileStatement))
            statementList.add(whileStatement);
      }
      //handling of all other types of statements in a similar manner
   }
}

public interface StatementInstanceChecker {
   public boolean instanceOf(Statement statement);
}

public class InstanceOfIfStatement implements
            StatementInstanceChecker {
   public boolean instanceOf(Statement statement) {
      if(statement instanceof IfStatement)
         return true;
      return false;
   }
}
```

Figure 6.8: Collection of *IfStatement* nodes existing within a given statement.

As it can be observed from Figure 6.8, method `getStatements()` in class `StatementCollector` invokes recursively itself in the branches of the *if/else if* structure corresponding to statements having a body, by passing as argument the statement corresponding to their body. In this way it is ensured that the AST structure will be completely traversed until the level of leaf statements not having a body. Furthermore, method `instanceOf()` is invoked through reference `instanceChecker` at the end of each branch, in order to check whether the statement in the given branch corresponds to the AST node type of interest that should be collected. The aforementioned check is achieved by means of polymorphism. The type of reference `instanceChecker` corresponds to interface `StatementInstanceChecker` which declares abstract method `instanceOf(Statement statement)`. The classes implementing this interface, such as class `InstanceOfIfStatement`, implement method `instanceOf()` by examining whether the statement of the parameter is an instance of the AST node type of interest (e.g., in the case of class `InstanceOfIfStatement`, it is examined whether the statement of the parameter is an instance of the *IfStatement* AST node type). In order to employ this mechanism and collect the AST nodes having a specific type, reference `instanceChecker` should be initialized with the appropriate `StatementInstanceChecker` subclass and next method `getStatements()` should be invoked, as exactly happens in the body of method `getIfS-`

tatements() where reference instanceChecker is initialized with a class instance creation of InstanceOfIfStatement subclass. Obviously, this infrastructure can be extended to support additional AST node types by adding appropriate StatementInstanceChecker subclasses, without modifying method getStatements() that performs the traversing of AST structures.

A similar infrastructure provides a mechanism for the collection of expressions having a specific type of interest. The major difference lies in the implementation of the AST structure traversing algorithm which extends the recursion depth until the level of leaf expressions (i.e., expressions not consisting of other expressions).

An alternative solution to this problem can be achieved by extending abstract class *ASTVisitor* of JDT Core API. Class *ASTVisitor* provides an abstract syntax tree traversal infrastructure based on the Visitor design pattern [39]. It contains a visit(*T* node) method for each concrete *ASTNode* type which returns a boolean value. If it returns a *true* value, the given node's child nodes will be visited next; however, if it returns a *false* value, the given node's child nodes will not be visited. By default, all visit(*T* node) methods simply return a *true* value. In the code of Figure 6.9, class IfStatementVisitor extends abstract class ASTVisitor and overrides method visit(IfStatement node) in order to collect all visited *IfStatements* into a list. The concrete visitor can be applied to an *ASTNode* reference by invoking method accept() through the *ASTNode* reference and passing as argument a class instance creation of the concrete visitor.

```
public class IfStatementVisitor extends ASTVisitor {
   private List<IfStatement> ifStatements =
            new ArrayList<IfStatement>();

   public boolean visit(IfStatement node) {
      ifStatements.add(node);
      return super.visit(node);
   }

   public List<IfStatement> getIfStatements() {
      return ifStatements;
   }
}
//get all if statements within a method declaration
private List<IfStatement> getIfStatements(MethodDeclaration method) {
   IfStatementVisitor ifStatementVisitor = new IfStatementVisitor();
   method.accept(ifStatementVisitor);
   return ifStatementVisitor.getIfStatements();
}
```

Figure 6.9: Collection of *IfStatement* nodes by extending abstract class *ASTVisitor*.

An important problem regarding the efficiency of *ASTVisitor* infrastructure is that by default it performs a complete traversal of the abstract syntax tree, regardless of the code level of the element being sought. For example, there is no need to visit all the expressions of an abstract syntax tree if a specific statement type is being looked for. In order to overcome this problem in the efficiency of the traversal, a set of appropriate visit() methods should be overridden in order to return a *false* value. For example, in the case of *IfStatements* the visit() methods corresponding to statements not having a body should be overridden to return a *false* value.

6.5 Finding the subclasses of a given class

In order to perform accurate static analysis of method calls, a polymorphic method call (i.e., an invocation of an abstract method) should be examined by analyzing the concrete methods implementing the invoked abstract method. This can be achieved by finding the subclasses extending (or implementing) the class (or interface) in which the abstract method is declared and then by searching within the subclasses for methods having the same signature with the abstract method.

Eclipse JDT Core provides a powerful index-based search infrastructure that can be used for various purposes, such as searching for references of a given type, method or field, finding the declaration of a given type, method or field, and finding the subclasses of a given class or interface. The code of Figure 6.10 employs the JDT Core search API in order to collect the subclass types of a given abstract class or interface type.

```
private Set<IType> getSubTypes(IType superType) {
   LinkedHashSet<IType> subTypes = new LinkedHashSet<IType>();
   try {
      SearchPattern searchPattern =
            SearchPattern.createPattern(superType,
            IJavaSearchConstants.IMPLEMENTORS);
      SearchEngine searchEngine = new SearchEngine();
      IJavaSearchScope scope =
            SearchEngine.createHierarchyScope(superType);
      SearchRequestor requestor = new TypeSearchRequestor(subTypes);

      searchEngine.search(searchPattern,
            new SearchParticipant[]
            {SearchEngine.getDefaultSearchParticipant()},
            scope, requestor, null);
   } catch (JavaModelException e) {
      e.printStackTrace();
   } catch (CoreException e) {
      e.printStackTrace();
   }
   return subTypes;
}

public class TypeSearchRequestor extends SearchRequestor {
   private Set<IType> subTypes;

   public TypeSearchRequestor(Set<IType> subTypes) {
      this.subTypes = subTypes;
   }

   public void acceptSearchMatch(SearchMatch match)
            throws CoreException {
      Object element = match.getElement();
      if (element instanceof IType) {
         subTypes.add((IType)element);
      }
   }
}
```

Figure 6.10: Collection of the subclass types for a given abstract type.

As it can be observed from Figure 6.10, the invocation of method `search()` through reference `searchEngine` inside the body of method `getSubTypes()` requires three basic arguments to be provided as input in order to operate.

The first argument concerns the search pattern. A search pattern can be created through two static methods named `createPattern()` which are provided by class `SearchPattern`.

1. The first static method, namely `createPattern(IJavaElement element, int limitTo)`, creates a search pattern from a Java element (parameter `element`) and parameter `limitTo` which determines the nature of the expected matches and can take one of the following values:
   - `IJavaSearchConstants.DECLARATIONS`: will search for declarations matching with the corresponding Java element. In case the Java element is a method, declarations of matching methods in subtypes will also be found.
   - `IJavaSearchConstants.REFERENCES`: will search for references to the given element.
   - `IJavaSearchConstants.ALL_OCCURRENCES`: will search for either declarations or references as specified above.
   - `IJavaSearchConstants.IMPLEMENTORS`: will search for types extending or implementing the abstract class or interface corresponding to the Java element.
2. The second static method, namely `createPattern(String stringPattern, int searchFor, int limitTo, int matchRule)`, creates a search pattern from a string pattern which may contain '*' or '?' wildcards. The parameter `searchFor` determines the nature of the searched elements and can take one of the following values:
   - `IJavaSearchConstants.CLASS`: looks only for classes.
   - `IJavaSearchConstants.INTERFACE`: looks only for interfaces.
   - `IJavaSearchConstants.TYPE`: looks for both classes and interfaces.
   - `IJavaSearchConstants.FIELD`: looks for fields.
   - `IJavaSearchConstants.METHOD`: looks for methods.
   - `IJavaSearchConstants.CONSTRUCTOR`: looks for constructors.
   - `IJavaSearchConstants.PACKAGE`: looks for packages.

   Finally, the parameter `matchRule` determines the matching rule of the search and can take one of the following values:
   - `SearchPattern.R_EXACT_MATCH`: the search pattern matches exactly the search result.
   - `SearchPattern.R_PATTERN_MATCH`: the search pattern contains one or more wildcards ('*' or '?').
   - `SearchPattern.R_PREFIX_MATCH`: the search pattern is a prefix of the search result.
   - `SearchPattern.R_REGEXP_MATCH`: the search pattern contains a regular expression.
   - `SearchPattern.R_CASE_SENSITIVE`: the search pattern matches the search result only if cases are the same.

In the code of Figure 6.10, static method `createPattern(IJavaElement element, int limitTo)` is used in order to create a search pattern for types extending or implementing the abstract class or interface corresponding to parameter `superType`.

The second argument that should be provided to method `search()` concerns the scope of the search. Setting of scope is very important, since it allows to restrict the search within certain Java elements instead of using the entire workspace as search space. A search scope can be created through two static methods which are provided by class `SearchPattern`. The first static method, namely `createJavaSearchScope(IJavaElement[] elements)`, creates a search scope limited to the Java elements specified by parameter `elements`. The second static method, namely `createHierarchyScope(IType type)`, creates a search scope limited to the hierarchy of the type specified by parameter `type`. In the code of Figure 6.10, the second static method is used in order to create a search scope limited to the hierarchy of parameter `superType`.

Finally, the last argument that should be provided to method `search()` concerns the search requestor. The search requestor is responsible for collecting the results from a search engine query. To create a search requestor, one should extend abstract class `SearchRequestor` and provide implementation for abstract method `acceptSearchMatch(SearchMatch match)` which is called after each search match. In the code of Figure 6.10, class `TypeSearchRequestor` extends abstract class `SearchRequestor` and in method `acceptSearchMatch()` checks whether the search match corresponds to an *IType* element and stores the identified subclass type into set `subTypes`.

6.6 Implementing refactoring transformations on Java source code

JDT Core provides two approaches in order to perform changes on abstract syntax trees. The first approach allows the direct modification of *ASTNodes*, while the second approach makes use of the *ASTRewrite* API which allows to record the change modifications intended to be performed on a given abstract syntax tree without affecting its original structure. The second approach is more suitable for the cases where the change modifications should be previewed without necessarily being applied on AST. In the case of JDeodorant, it is very common that an abstract syntax tree (i.e., *CompilationUnit*) presents more than one identified refactoring opportunities. As a result, the user should be able to inspect the change modifications corresponding to each refactoring in a consecutive manner without actually applying the refactorings on source code. In order to achieve this feature, the original AST should remain intact after the preview of the change modifications imposed by each refactoring. Consequently, JDeodorant adopts the approach based on *ASTRewrite* API to perform the required change modifications.

An instance of *ASTRewrite* for a specific AST can be obtained by invoking static method `create(AST ast)` which is provided by class *ASTRewrite*. This method requires as parameter an instance of *AST* type which can be obtained from any *ASTNode* by invoking method `getAST()`. An *AST* instance serves as the common owner of the *ASTNodes* belonging to a given abstract syntax tree, and as the factory for creating new *ASTNodes* owned by that instance. Class *ASTRewrite* provides four basic methods in order to record change modifications:

- `set(ASTNode node, StructuralPropertyDescriptor property, Object value, TextEditGroup editGroup)` which sets the given property of the given node with the given value.
- `getListRewrite(ASTNode node, ChildListPropertyDescriptor property)` which creates and returns a list rewriter (*ListRewrite*) for describing modifications to the given list property of the given node.
- `replace(ASTNode node, ASTNode replacement, TextEditGroup editGroup)` which replaces the given node with the given replacement node.
- `remove(ASTNode node, TextEditGroup editGroup)` which removes the given node from its parent node.

As it can be observed, the first two methods can be used to modify the structural properties of the `node` parameter and require a *StructuralPropertyDescriptor* as parameter, while the last two methods affect the `node` parameter itself. The structural properties of *ASTNodes* are grouped into three different kinds, namely properties holding values which are not *ASTNodes* (i.e., primitive types such as *int* and *boolean* or simple types such as strings and operators), properties which contain a single child *ASTNode*, and properties which contain a list of child *ASTNodes*. Accordingly, abstract class *StructuralPropertyDescriptor* has three concrete subclasses, namely *SimplePropertyDescriptor*, *ChildPropertyDescriptor* and *ChildListPropertyDescriptor*, which serve as property descriptors for the aforementioned kinds of properties, respectively. The code of Figure 6.11 creates a new *MethodDeclaration* by assigning a name, a return type, an access modifier, parameters and a method body to it and finally adds the new *MethodDeclaration* to an already existing *TypeDeclaration*. All modifications are performed through *ASTRewrite* and *ListRewrite* operations.

```
AST ast = typeDeclaration.getAST();
ASTRewrite rewriter = ASTRewrite.create(ast);
MethodDeclaration newMethod = ast.newMethodDeclaration();

SimpleName methodName = ast.newSimpleName(...);
rewriter.set(newMethod, MethodDeclaration.NAME_PROPERTY,
      methodName, null);

PrimitiveType returnType = ast.newPrimitiveType(PrimitiveType.VOID);
rewriter.set(newMethod, MethodDeclaration.RETURN_TYPE2_PROPERTY,
      returnType, null);

ListRewrite modifierRewrite = rewriter.getListRewrite(newMethod,
      MethodDeclaration.MODIFIERS2_PROPERTY);
Modifier accessModifier =
      ast.newModifier(Modifier.ModifierKeyword.PRIVATE_KEYWORD);
modifierRewrite.insertLast(accessModifier, null);

ListRewrite parameterRewrite = rewriter.getListRewrite(newMethod,
      MethodDeclaration.PARAMETERS_PROPERTY);
SingleVariableDeclaration parameter =
      ast.newSingleVariableDeclaration();
SimpleName parameterName = ast.newSimpleName(...);
rewriter.set(parameter, SingleVariableDeclaration.NAME_PROPERTY,
      parameterName, null);
Type parameterType = ast.newSimpleType(ast.newSimpleName(...));
rewriter.set(parameter, SingleVariableDeclaration.TYPE_PROPERTY,
      parameterType, null);
parameterRewrite.insertLast(parameter, null);
```

```
Block methodBody = ast.newBlock();
ListRewrite methodBodyRewrite = rewriter.getListRewrite(methodBody,
      Block.STATEMENTS_PROPERTY);
...
rewriter.set(newMethod, MethodDeclaration.BODY_PROPERTY,
      methodBody, null);

ListRewrite bodyDeclarationRewrite =
      rewriter.getListRewrite(typeDeclaration,
      TypeDeclaration.BODY_DECLARATIONS_PROPERTY);
bodyDeclarationRewrite.insertLast(newMethod, null);

try {
   TextEdit edit = rewriter.rewriteAST();
} catch (JavaModelException e) {
   e.printStackTrace();
}
```

Figure 6.11: Creation of a new *MethodDeclaration* AST node.

The final try block in Figure 6.11, invokes method `rewriteAST()` in order to convert all modifications recorded by the `rewriter` into a *TextEdit* object representing the corresponding text edits to the source of the *ICompilationUnit* from which the AST was created from. The resulting *TextEdit* can be applied to the *IDocument* (corresponding to the *ICompilationUnit*) in order to actually transfer the changes on source code. This is achieved by invoking method `apply(IDocument document)` of class *TextEdit* which returns an *UndoEdit* object that can be used to revert the source code in its original status. Class *UndoEdit* is a subclass of *TextEdit* that encapsulates the reverse changes of an applied *TextEdit*.

Instead of manually applying the *TextEdits* resulting from the description of modifications through *ASTRewrites*, Eclipse provides the Refactoring Language Toolkit (LTK) infrastructure that allows the integration of a refactoring transformation with the refactoring history (i.e., undo and redo operations in Eclipse workbench) and the utilization of refactoring wizards enabling the automatic preview of the change modifications imposed by a refactoring as well as the determination of parameters required for the application of a refactoring through user input pages.

The first step in order to employ the refactoring infrastructure provided by Eclipse is to extend the abstract superclass *Refactoring* which declares four abstract methods that should be implemented by its subclasses. These abstract methods are the following:

- `checkInitialConditions(IProgressMonitor pm)`: checks some initial conditions that should be valid before the application of the refactoring and returns a *RefactoringStatus* object representing the outcome of condition checking. The latter object maintains a list of status entries (*RefactoringStatusEntry* objects) which describe particular problems detected during condition checking. Each status entry may have a different severity level. Severity levels are ordered as OK < INFO < WARNING < ERROR < FATAL. The severity of RefactoringStatus is the maximum of the severities of its status entries. This method can also be used to compute some initial values for refactoring-related parameters that should be presented to the user.
- `checkFinalConditions(IProgressMonitor pm)`: checks any remaining conditions that should be valid after the user has provided all input necessary

to perform the refactoring and returns a *RefactoringStatus* object. It is always called after the execution of method `checkInitialConditions()`. This method is not intended for validating the user input. Instead, the user input should be validated directly on the input page of the refactoring wizard.

- `getName()`: returns the name of the refactoring.
- `createChange(IProgressMonitor pm)`: creates a *Change* object that performs the actual workspace transformation.

Abstract class *Change* has several subclasses representing various types of changes that may occur in the Eclipse workbench. In the case of refactorings on Java source code the most suitable type of *Change* is *CompilationUnitChange* which is a special *TextFileChange* that operates on an *ICompilationUnit* in the workspace. In order to create an operational *CompilationUnitChange* object, the *TextEdit* resulting from a single *ASTRewrite* or the *MultiTextEdit* resulting from the aggregation of multiple *TextEdits* resulting from different *ASTRewrites* should be set to the *CompilationUnitChange* through method `setEdit(TextEdit edit)`. Additionally, all subtypes of *TextChange* (such as *CompilationUnitChange*) allow to define *TextEditGroups* through method `addTextEditGroup(TextEditGroup group)` in order to present parts of the change modifications as different preview groups. This allows to preview the changes in a *CompilationUnit* as multiple groups of text modifications and not as a single text modification covering the entire *CompilationUnit*. Figure 6.12 shows a case of Extract Method refactoring where the change modifications occurring within the body of the original method are captured by a different preview group compared to the group of change modifications that create the extracted method. Obviously, in this way the user can more easily discriminate the change modifications imposed by complex refactoring transformations.

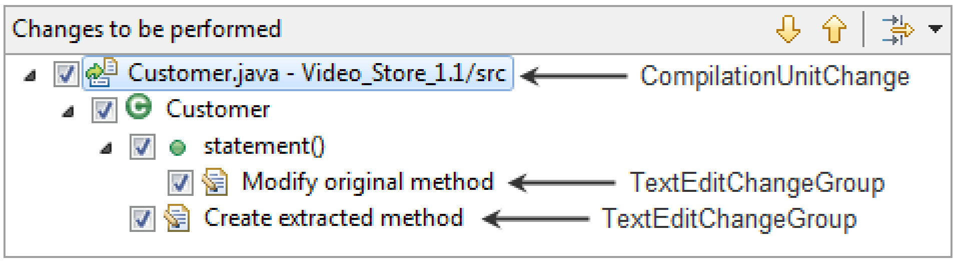

Changes to be performed
Customer.java - Video_Store_1.1/src ← CompilationUnitChange
Customer
statement()
Modify original method ← TextEditChangeGroup
Create extracted method ← TextEditChangeGroup

Figure 6.12: Representation of *TextEditChangeGroups* in a *CompilationUnitChange*.

The code of Figure 6.13 contains a typical implementation of method `createChange()` for a single *CompilationUnitChange* that can be easily extended for refactorings affecting more than one *CompilationUnits*. The final part of this implementation creates a *CompositeChange* object which facilitates the composition of multiple *Changes* into a single one. The created *CompositeChange* object should override method `getDescriptor()` to return an instance of *RefactoringChangeDescriptor* which in turn encapsulates a *RefactoringDescriptor* instance. A refactoring descriptor contains refactoring-specific data which allows the framework to completely reconstruct a particular refactoring instance and execute it on an arbitrary workspace. In other words, a refactoring descriptor enables the execution of a redo operation for a refactoring which has been undone in the refactoring history of the Eclipse workbench.

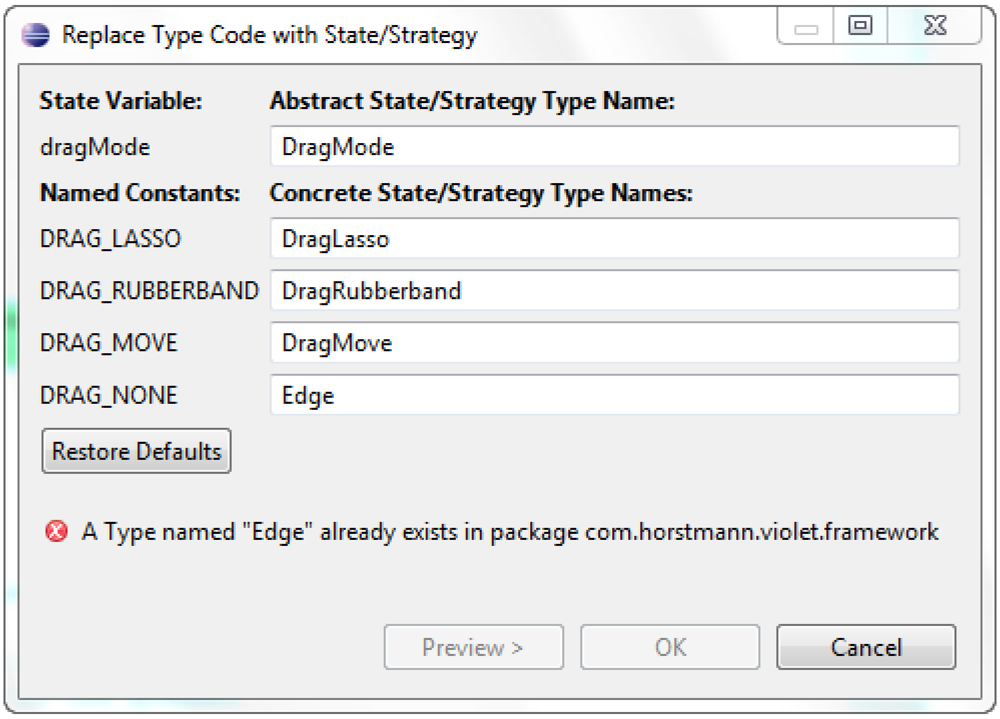
```
public Change createChange(IProgressMonitor pm) throws CoreException,
            OperationCanceledException {
   try {
      pm.beginTask("Creating change...", 1);
      TextEdit edit = rewriter.rewriteAST();
      MultiTextEdit root = new MultiTextEdit();
      root.addChild(edit);
      //
      ICompilationUnit iCompilationUnit =
         (ICompilationUnit)compilationUnit.getJavaElement();
      CompilationUnitChange compilationUnitChange =
         new CompilationUnitChange("", iCompilationUnit);
      //
      TextEditGroup group = new TextEditGroup("Create new method",
         new TextEdit[] {edit});
      compilationUnitChange.addTextEditGroup(group);
      compilationUnitChange.setEdit(root);

      CompositeChange change = new CompositeChange(getName(),
         new Change[] {compilationUnitChange} ) {

         public ChangeDescriptor getDescriptor() {
            return new RefactoringChangeDescriptor(...);
         }
      };
      return change;
   } catch (JavaModelException e) {
      e.printStackTrace();
   } finally {
      pm.done();
   }
}
```

Figure 6.13: Implementation of abstract method `createChange()` for a single *CompilationUnitChange*.

Eclipse LTK offers the ability to collect user input required for the application of the refactoring and preview the change modifications imposed by the refactoring through a special wizard for refactorings which can be utilized by extending abstract class *RefactoringWizard*. The subclass extending *RefactoringWizard* will automatically provide a change preview page where change modifications can be previewed before the application of the refactoring. Additional user input pages can be added to the refactoring wizard by implementing abstract method `addUserInputPages()` and invoking method `addPage(IWizardPage page)` in order to add subclasses of abstract class *UserInputWizardPage* as user input pages. Figure 6.14 shows an example of user input page in the *Replace Type Code with State/Strategy* refactoring wizard. This user input page provides a set of default names for the classes of the State/Strategy design pattern that will be introduced by the refactoring, based on the names of the variable holding the current state and the named constants representing all possible states. The user has the ability to rename the default names, while at the same time the user input is validated against a set of preconditions. An example of a precondition being examined is that the input values should not have the same name with already existing types in the given Java project. The refactoring wizard allows the user to preview or apply the refactoring (by pressing the Preview or OK button, respectively) only if all preconditions are satisfied.



State Variable:	Abstract State/Strategy Type Name:
dragMode	DragMode
Named Constants:	**Concrete State/Strategy Type Names:**
DRAG_LASSO	DragLasso
DRAG_RUBBERBAND	DragRubberband
DRAG_MOVE	DragMove
DRAG_NONE	Edge

A Type named "Edge" already exists in package com.horstmann.violet.framework

Figure 6.14: An example of user input page in the *Replace Type Code with State/Strategy* refactoring wizard.

Chapter 7

7 Conclusions and Future Work

This work presented methods and techniques providing a concrete solution for three major design problems in object-oriented systems, namely *Feature Envy* [36] which constitutes a sign of violating the design principle indicating that behavior (i.e., methods) and related data (i.e., attributes) should be grouped together in the same class, *State or Type Checking* [36, 54, 26] bad smell which constitutes a direct violation of the *Open/Closed* principle stating that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification, and *Long Method* [36] bad smell which constitutes a violation of the principle indicating that methods should be as small, simple and cohesive as possible.

The major contribution of this work is that it faces the problem of improving the design quality of an object-oriented system by identifying refactoring opportunities which resolve bad smells existing in source code. In this way, it is possible to provide solutions that not only detect given bad smells, but can also resolve them by appropriate refactorings. Furthermore, this refactoring-oriented approach allows to examine whether the resulting refactoring solutions are feasible and behavior-preserving and assess their impact on certain aspects of design quality.

7.1 Discussion of evaluation results

Regarding Feature Envy bad smell, a method that identifies *Move Method* [36] refactoring opportunities as a means to resolve Feature Envy problems existing within an object-oriented system has been developed. The qualitative analysis of the refactoring suggestions extracted for an open-source project revealed that they can be useful in assisting the designer to improve design quality. The study of coupling and cohesion evolution when successively applying the refactoring solutions extracted for two open-source projects has shown that the refactorings suggested by the proposed method have a positive impact on both coupling and cohesion. The assessment by an independent designer of the refactoring suggestions extracted for a system that he developed indicated that the proposed method is capable of producing conceptually sound suggestions. Finally, CPU time measurements have shown that the efficiency of the approach depends primarily on the number of the extracted refactoring suggestions and secondly on the size of the system under study.

Regarding State Checking bad smell, a technique that identifies *Replace Type Code with State/Strategy* and *Replace Conditional with Polymorphism* [36] refactoring opportunities as a means to resolve state-check and RTTI problems, respectively, has been developed. The comparison of the refactoring opportunities identified by an independent expert on three open-source projects with the results of the proposed technique has shown a moderate precision and relatively high recall. The small number of false negatives encourages the use of the proposed technique as a semi-automatic approach, where the designer eventually decides whether a suggested refactoring should be applied or not. A second experiment investigated by means of binary logistic regression the correlation between three quantitative factors and the decision of the expert. The results indicate that the designer agrees in introducing polymorphism when an inheritance hierarchy is extensively utilized by adding several polymorphic methods, when a large number of statements are moved from the conditional structures to the concrete State subclasses and when the number of concrete State subclasses (representing all possible states that an object may obtain for a given property) being created is relatively small. Finally, performance analysis has shown that the efficiency of the proposed technique in terms of computation time depends on the total number of conditional structures found in the system under examination.

Regarding Long Method bad smell, a method that identifies *Extract Method* [36] refactoring opportunities as a means to resolve Long Method problems existing within an object-oriented system has been developed. The evaluation has shown that the proposed method is able to capture slices of code implementing a distinct and independent functionality compared to the rest of the original method and thus lead to extracted methods with useful functionality. At the same time, the identified refactoring opportunities can help significantly to resolve existing design flaws by decomposing complex methods, removing duplicated code among several methods and extracting code fragments suffering from Feature Envy. Furthermore, the identified refactoring opportunities have a positive impact on the cohesion of the decomposed methods and lead to highly cohesive extracted methods. Finally, the defined behavior preservation rules can successfully exclude refactoring opportunities that could possibly cause a change in program behavior.

7.2 General Conclusions

A general conclusion drawn from this work is that each design problem has its own specific structural diversities and should be handled in a distinct manner. As a result, it is rather difficult and almost impossible to develop a generic method that takes into account all structural aspects of design problems in a way that is able to efficiently produce high quality solutions for any type of problem.

Another important conclusion is that an approach aiming at improving the design quality of an existing system should encompass human judgement in order to take into account conceptual issues which cannot be inferred from purely structure-based analysis. As a result, a semi-automatic approach that produces a list of ranked solutions (based on structural characteristics of the program under examination) which are eventually approved or disapproved by the designer of the program seems to be the most suitable approach for improving the design quality of system.

7.3 Future Research

The developed methods (i.e., methods aiming at identifying refactoring opportunities) can be combined along with methods aiming at detecting refactoring activities that occurred in past versions of a software system [111, 112, 113] in order to perform empirical studies investigating the refactoring habits of developers and maintainers. In this way, it will be possible to answer some important research questions, such as:

- The refactoring activities which are actually performed by developers and maintainers target at removing existing design problems or serve for other purposes?
- Is there a systematic effort to remove design problems through refactorings (i.e., targeted preventive maintenance) during maintenance or do refactoring activities have a sporadic nature?
- Is it observed an increased maintainability to parts of a program which have been refactored due to the existence of a design problem?

Another interesting research perspective is the investigation and development of a method that models dependencies among different types of refactoring opportunities. In this way, it will be possible to determine a sequence refactoring applications which ensures that no conflicts will exist among different types of refactorings (i.e., the application of a given refactoring does not hinder or make infeasible the application of another).

Bibliography

[1] A. Abadi, R. Ettinger, and Y. A. Feldman, "Re-Approaching the Refactoring Rubicon," *2nd ACM Workshop on Refactoring Tools* (WRT'08), 2008.

[2] M. Aeschlimann, D. Bäumer, and J. Lanneluc, "Java Tool Smithing - Extending the Eclipse Java Development Tools," *eclipseCON*, 2005.

[3] V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[4] A. Ananda Rao, and K. Narendar Reddy, "Detecting Bad Smells in Object Oriented Design Using Design Change Propagation Probability Matrix," *International MultiConference of Engineers and Computer Scientists 2008 Vol I* (IMECS'08), pp. 1001-1007, 2008.

[5] E. Arisholm, and D. I.K. Sjøberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 521-534, 2004.

[6] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Costs," *Communications of the ACM*, vol. 36, no. 11, pp. 81-94, November 1993.

[7] J. Bansiya, and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4-17, January 2002.

[8] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, October 1996.

[9] J.-F. Bergeretti, and B.A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 37-61, 1985.

[10] D. Binkley, and K. B. Gallagher, "Program Slicing," *Advances in Computers*, vol.43, 1996.

[11] A. B. Binkley, and S. R. Schach, "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures," *20th International Conference on Software Engineering* (ICSE'98), pp. 452-455, 1998.

[12] L. C. Briand, J. W. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65-117, 1998.

[13] L. C. Briand, J. W. Daly, and J. K. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, January/February 1999, pp. 91-121.

[14] L. C. Briand, J. Wüst, S.V. Ikonomovski, and H. Lounis, "Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study," *21st International Conference on Software Engineering* (ICSE'99), pp. 345-354, 1999.

[15] L. C. Briand, J. Wüst, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," *International Conference on Software Maintenance* (ICSM'99), pp. 475-482, 1999.

[16] L. C. Briand, and J. Wüst, "Modeling Development Effort in Object-Oriented Systems Using Design Properties," *IEEE Transactions on Software Engineering*, vol. 27, no. 11, pp. 963-986, November 2001.

[17] F. Brito e Abreu, "The MOOD Metrics Set," *9th European Conference on Object-Oriented Programming* (ECOOP'95), Workshop on Metrics, 1995.

[18] F. Brito e Abreu, and W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality," *3rd International Software Metrics Symposium* (METRICS'96), pp.90-99, 1996.

[19] W. J. Brown, R. C. Malveau, H.W. McCormick III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, 1998.

[20] M. A. Chaumun, H. Kabaili, R. K. Keller, F. Lustman, and G. Saint-Denis, "Design Properties and Object-Oriented Software Changeability," *4th European Conference on Software Maintenance* (CSMR'2000), pp. 45-54, 2000.

[21] S. R. Chidamber, and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, June 1994, pp. 476-493.

[22] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 629-639, August 1998.

[23] A. Cimitile, A. De Lucia, and M. Munro, "A Specification Driven Slicing Process for Identifying Reusable Functions," *Software Maintenance: Research and Practice*, vol. 8, pp. 145-178, 1996.

[24] M. Day, R. Gruber, B. Liskov, and A. C. Myers, "Subtypes vs. Where Clauses: Constraining Parametric Polymorphism," *10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA'95), pp. 156-168, 1995.

[25] A. De Lucia, M. Harman, R. Hierons, and J. Krinke, "Unions of Slices are not Slices," *7th European Conference on Software Maintenance and Reengineering* (CSMR'03), pp. 363-367, 2003.

[26] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*, Morgan Kaufman, 2003.

[27] S. Demeyer, "Refactor Conditionals into Polymorphism: What's the Performance Cost of Introducing Virtual Calls?," *21st IEEE International Conference on Software Maintenance* (ICSM'05), pp. 627-630, 2005.

[28] S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall and M. El-Ramly, "The LAN-simulation: A Refactoring Teaching Example," *8th In-*

ternational Workshop on Principles of Software Evolution (IWPSE'05), pp. 123-134, 2005.

[29] K. Dhambri, H. Sahraoui, and P. Poulin, "Visual Detection of Design Anomalies," *12th European Conference on Software Maintenance and Reengineering* (CSMR'08), pp. 279-283, 2008.

[30] B. Du Bois, "A Study of Quality Improvements by Refactoring," Ph.D. dissertation, University of Antwerp, 2006.

[31] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring - Improving Coupling and Cohesion of Existing Code," *11th Working Conference on Reverse Engineering* (WCRE'04), pp. 144-151, 2004.

[32] Eclipse Corner Article: Abstract Syntax Tree, Available at: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html

[33] Eclipse Java development tools (JDT) Overview, Available at: http://www.eclipse.org/jdt/overview.php

[34] R. Ettinger, "Refactoring via Program Slicing and Sliding," Ph.D. dissertation, University of Oxford, United Kingdom, 2007.

[35] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July 1987.

[36] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, Boston, MA, 1999.

[37] A. Frazer, "Reverse engineering - Hype, Hope or Here?," in *Software Reuse and Reverse Engineering in Practice*, P. A. V. Hall, Ed. Chapman & Hall, 1992, pp. 209-243.

[38] K. B. Gallagher, and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751-761, August 1991.

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

[40] C. Ghezzi, M. Jazayeri and D. Mandrioli, *Fundamentals of Software Engineering*, Second Edition, Prentice Hall, Upper Saddle River, NJ, 2003.

[41] G. K. Gill, and C. F. Kemerer, "Cyclomatic Complexity Density and Software Maintenance Productivity," *IEEE Transactions on Software Engineering*, vol. 17, no. 12, pp. 1284-1288, December 1991.

[42] M. Harman, and R. M. Hierons, "An Overview of Program Slicing," *Software Focus*, vol. 2, no. 3, pp. 85-92, 2001.

[43] M. Harman, D. Binkley, and S. Danicic, "Amorphous Program Slicing," *Journal of Systems and Software*, vol. 68, no. 1, pp. 45-64, 2003.

[44] M. Harman, D. Binkley, R. Singh, and R. M. Hierons, "Amorphous Procedure Extraction," *4th IEEE International Workshop on Source Code Analysis and Manipulation* (SCAM'04), pp. 85-94, 2004.

[45] M. Harman, and L. Tratt, "Pareto Optimal Search Based Refactoring at the Design Level," *9th Annual Genetic and Evolutionary Computation Conference* (GECCO'07), pp. 1106-1113, 2007.

[46] M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems," *International Symposium on Applied Corporate Computing* (ISACC'95), 1995.

[47] C. S. Horstmann, *Object-Oriented Design and Patterns*, Second Edition, Wiley, 2006.

[48] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, 1990.

[49] JDeodorant website, http://www.jdeodorant.com

[50] T. Jiang, M. Harman, Y. Hassoun, "Analysis of Procedure Splitability," *15th Working Conference on Reverse Engineering* (WCRE'08), pp. 247-256, 2008.

[51] B.-K. Kang, and J. M. Bieman, "Using design abstractions to visualize, quantify, and restructure software," *The Journal of Systems and Software*, vol. 42, pp. 175-187, 1998.

[52] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, "Automated Support for Program Refactoring using Invariants," 17th *IEEE International Conference on Software Maintenance* (ICSM'01), pp. 736-743, 2001.

[53] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," *18th IEEE International Conference on Software Maintenance* (ICSM'02), pp. 576-585, 2002.

[54] J. Kerievsky, *Refactoring to Patterns*, Addison Wesley, 2004.

[55] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells," *9th International Conference on Quality Software* (QSIC'09), pp. 305-314, 2009.

[56] R. Komondoor, and S. Horwitz, "Effective, Automatic Procedure Extraction," *11th IEEE International Workshop on Program Comprehension* (IWPC'03), 2003.

[57] B. Korel, and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155-163, 1988.

[58] T. Kosar, M. Mernik, V. Žumer, "JART: Grammar-Based Approach to Refactoring," *28th Annual International Computer Software and Applications Conference* (COMPSAC'04), pp. 502-507, 2004.

[59] S. Kumar, and S. Horwitz, "Better Slicing of Programs with Jumps and Switches," *5th International Conference on Fundamental Approaches to Software Engineering* (FASE'02), pp. 96- 112, 2002.

[60] A. Lakhotia, and J.-C. Deprez, "Restructuring Programs by Tucking Statements into Functions," *Information and Software Technology*, vol. 40, no. 11-12, pp. 677-690, 1998.

[61] W. Landi, B. G. Ryder, and S. Zhang, "Interprocedural Modification Side Effect Analysis with Pointer Aliasing," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'93), pp. 56-67, 1993.

[62] F. Lanubile, and G. Visaggio, "Extracting Reusable Functions by Flow Graph-Based Program Slicing," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 246-259, April 1997.

[63] L. Larsen, and M. J. Harrold, "Slicing object-oriented software," *International Conference on Software Engineering* (ICSE'96), pp. 495-505, 1996.

[64] W. Li, and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111-122, 1993.

[65] D. Liang, and M. J. Harrold, "Slicing Objects Using System Dependence Graphs," *14th IEEE International Conference on Software Maintenance* (ICSM'98), pp. 358-367, 1998.

[66] B. P. Lientz, and E. B. Swanson, *Software maintenance management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, Addison-Wesley, Boston, MA, USA, 1980.

[67] O. Maqbool, and H.A. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759-780, November 2007.

[68] J. Martin, and C. L. McClure, *Software Maintenance: The Problems and its Solutions*, Prentice Hall, 1983.

[69] R. C. Martin, *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2003.

[70] K. Maruyama, "Automated Method-Extraction Refactoring by Using Block-Based Slicing," *Symposium on Software Reusability* (SSR'01), pp.31-40, 2001.

[71] T. Mens, and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126-139, February 2004.

[72] T. M. Meyers and D. Binkley, "An Empirical Study of Slice-Based Cohesion and Coupling Metrics," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 1, December 2007.

[73] B. S. Mitchell, and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193-208, March 2006.

[74] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20-36, Jan./Feb. 2010.

[75] G. C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?," *IEEE Software*, vol. 23, no. 4, pp. 76-83, July 2006.

[76] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *31st International Conference on Software Engineering* (ICSE'09), 2009.

[77] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu, "Work Experience versus Refactoring to Design Patterns: A Controlled Experiment," *14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE'06), pp. 12-22, 2006.

[78] T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu, "Do Maintainers Utilize Deployed Design Patterns Effectively?," *29th International Conference on Software Engineering* (ICSE'07), pp. 168-177, 2007.

[79] J. T. Nosek, and P. Palvia, "Software Maintenance Management: Changes in the Last Decade," *Journal of Software Maintenance: Research and Practice*, vol. 2, no. 3, pp. 157-174, 1990.

[80] M. Ó Cinnéide, and P. Nixon, "A Methodology for the Automated Introduction of Design Patterns," *15th IEEE International Conference on Software Maintenance* (ICSM'99), pp. 463-472, 1999.

[81] M. Ó Cinnéide, "Automated Application of Design Patterns: A Refactoring Approach," Ph.D. dissertation, University of Dublin, Trinity College, 2000.

[82] M. O'Keeffe, and M. Ó Cinnéide, "Search-based software maintenance," *10th European Conference on Software Maintenance and Reengineering* (CSMR'06), pp. 249-260, 2006.

[83] M. O'Keeffe, and M. Ó Cinnéide, "Getting the Most from Search-Based Refactoring," *9th Annual Genetic and Evolutionary Computation Conference* (GECCO'07), pp. 1114-1120, 2007.

[84] M. O'Keeffe, and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *The Journal of Systems and Software*, vol. 81, no. 4, pp. 502-516, April 2008.

[85] F. Ohata, and K. Inoue, "JAAT: Java Alias Analysis Tool for Program Maintenance Activities," *9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing* (ISORC'06), pp. 232-244, 2006.

[86] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[87] D. L. Parnas, "Software aging," *16th International Conference on Software Engineering* (ICSE'94), pp. 279-287, 1994.

[88] C. Parnin, C. Görg, and O. Nnadi, "A Catalogue of Lightweight Visualizations to Support Code Smell Inspection," *4th ACM symposium on Software visualization* (SOFTVIZ'08), pp. 77-86, 2008.

[89] T. M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, Wiley, New York, 1996.

[90] M. Polo, M. Piattini, and F. Ruiz, "A Methodology for Software Maintenance," in *Advances in Software Maintenance Management: Technologies and Solutions*, Idea Group Publishing, Idea Group Inc., 2003, pp. 228-254.

[91] L. Prechelt, B. Unger, W.F. Tichy, P. Brössler, and L.G. Votta, "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1134-1144, 2001.

[92] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, Third edition, McGraw-Hill, 1993.

[93] I. Refanidis, and A. Alexiadis, "SelfPlanner: An Intelligent Web-based Calendar Application," *17th International Conference on Automated Planning and Scheduling Systems* (ICAPS'07), 2007.

[94] A. J. Riel, *Object-oriented design heuristics*, Addison-Wesley, 1996.

[95] M. Salehie, S. Li, and L. Tahvildari, "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws," *14th IEEE International Conference on Program Comprehension* (ICPC'06), pp. 159-168, 2006.

[96] S. R. Schach, *Software engineering*, Richard D. Irwin/Aksen Associates, IL, USA, 1990.

[97] R. Seacord, D. Plakosh, and G. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*, SEI Series in Software Engineering, Addison-Wesley, 2003.

[98] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems," *8th Annual Conference on Genetic and Evolutionary Computation* (GECCO'06), pp. 1909-1916, 2006.

[99] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics Based Refactoring," *5th European Conference on Software Maintenance and Reengineering* (CSMR'01), pp. 30-38, 2001.

[100] L. Tahvildari, and K. Kontogiannis, "A Metric-Based Approach to Enhance Design Quality through Meta-pattern Transformations," *7th European Conference on Software Maintenance and Reengineering* (CSMR'03), pp. 183-192, 2003.

[101] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121-189, 1995.

[102] P. Tonella, "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 495-509, June 2003.

[103] T. Tourwé, and T. Mens, "Identifying Refactoring Opportunities Using Logic Meta Programming," *7th European Conference on Software Maintenance and Reengineering* (CSMR'03), pp. 91-100, 2003.

[104] A. Trifu, and R. Marinescu, "Diagnosing Design Problems in Object Oriented Systems," *12th Working Conference on Reverse Engineering* (WCRE'05), pp.155-164, 2005.

[105] A. Trifu, and U. Reupke, "Towards Automated Restructuring of Object Oriented Systems," *11th European Conference on Software Maintenance and Reengineering* (CSMR'07), pp.39-48, 2007.

[106] E. Van Emden, and L. Moonen, "Java Quality Assurance by Detecting Code Smells," *9th Working Conference on Reverse Engineering* (WCRE'02), pp. 97-106, 2002.

[107] H. Van Vliet, *Software Engineering: Principles and Practice*, Second Edition, Wiley, 2000.

[108] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352-357, 1984.

[109] P. Wendorff, "Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project," *5th European Conference on Software Maintenance and Reengineering* (CSMR'01), pp. 77-84, 2001.

[110] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.

[111] Z. Xing, and E. Stroulia, "Analyzing the evolutionary history of the logical design of object-oriented software," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 850-868, October 2005.

[112] Z. Xing, and E. Stroulia, "UMLDiff: An algorithm for object-oriented design differencing," *20th International Conference on Automated Software Engineering* (ASE'05), pp. 54-65, 2005.

[113] Z. Xing, and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries," *13th Working Conference on Reverse Engineering* (WCRE'06), pp. 263-274, 2006.

[114] L. M. Ott, and J. J. Thuss, "The Relationship between Slices and Module Cohesion," *11th International Conference on Software Engineering* (ICSE'89), pp. 198-204, 1989.

[115] L. M. Ott, and J. J. Thuss, "Slice-based metrics for estimating cohesion," *First International Software Metrics Symposium* (METRICS'93), pp. 71-81, 1993.

Appendix A

The following auxiliary functions examine whether a given method is an accessor or delegate method. The description of the functions assumes that such methods are written in a certain way, following the most common conventions.

```
(Field or null) isGetter(Method m) ≡
    size of m.parameters = 0 ∧ size of m.methodBody.statements = 1 ∧
    ∃ ReturnStatement r ∈ m.methodBody.statements where
    r.returnedExpression is VariableAccess v ∧ v.declaration is Field f ∧
    f.type = m.returnType
    return f
```

```
(Field or null) isSetter(Method m) ≡
    size of m.parameters = 1 ∧ size of m.methodBody.statements = 1 ∧
    ∃ AssignmentStatement a ∈ m.methodBody.statements where
    (a.assignment.leftHandSide is VariableAccess $v_1$ ∧ $v_1$.declaration is Field f) ∧
    (a.assignment.rightHandSide is VariableAccess $v_2$ ∧
    $v_2$.declaration = m.parameters[0]) ∧ f.type = m.parameters[0].type
    return f
```

```
(Type or null) elementTypeOfCollection(VariableDeclaration d) ≡
    if d.type is CollectionType cType ∧ cType.type ∈ {Collection, List, AbstractCollection, AbstractList, ArrayList, LinkedList, Vector,
    Set, AbstractSet, HashSet, LinkedHashSet, SortedSet, TreeSet}
    return cType.elementType
    else if d.type is MapType mType ∧ mType.type ∈ {Map, AbstractMap, HashMap, Hashtable, SortedMap, TreeMap, IdentityHashMap, WeakHashMap}
    return mType.valueType
```

* the element type is inferred by the generic type(s) of the field type, or by the type of the parameter of the collection setter methods corresponding to the field

```
(Field or null) isCollectionGetter(Method m) ≡
    (size of m.parameters = 0 ∧ size of m.methodBody.statements = 1 ∧
    ∃ ReturnStatement r ∈ m.methodBody.statements where
    r.returnedExpression is MethodInvocation methodInv ∧
    methodInv.invokeExpression is VariableAccess v ∧
    v.declaration is Field f ∧ elementTypeOfCollection(f) ≠ null ∧
    methodInv.name ∈ {iterator, toArray, listIterator,
    elements, keySet, entrySet, values}
    return f) ∨
    (size of m.parameters = 1 ∧ size of m.methodBody.statements = 1 ∧
    ∃ ReturnStatement r ∈ m.methodBody.statements where
    r.returnedExpression is MethodInvocation methodInv ∧
    methodInv.invokeExpression is VariableAccess v ∧ v.declaration is Field f ∧
    elementType = elementTypeOfCollection(f) ≠ null ∧
    methodInv.name ∈ {get, elementAt} ∧ elementType = m.returnType ∧
    positionOfArgument(methodInv, m.parameters[0]) ≠ -1
    return f)

(Field or null) isCollectionSetter(Method m) ≡
    size of m.parameters = 1 ∧ size of m.methodBody.statements = 1 ∧
    ∃ MethodInvocationStatement s ∈ m.methodBody.statements where
    s.methodInvocation.invokeExpression is VariableAccess v ∧
    v.declaration is Field f ∧ elementTypeOfCollection(f) ≠ null ∧
    s.methodInvocation.name ∈ {add, remove, addAll, removeAll,
    retainAll, addElement, removeElement, put} ∧
    positionOfArgument(s.methodInvocation, m.parameters[0]) ≠ -1
    return f

int positionOfArgument(MethodInvocation inv, Parameter param) ≡
    for i = 1 to size of inv.arguments
        if inv.arguments[i] is VariableAccess arg ∧ arg.declaration = param
            return i
    return -1

(Method or null) isDelegate(Method m) ≡
    (size of m.methodBody.statements = 1 ∧
    ∃ MethodInvocationStatement s ∈ m.methodBody.statements where
    s.methodInvocation.declaringClass.type ∈ program.classTypes ∧
    ((s.methodInvocation.invokeExpression is VariableAccess v ∧ v.declaration ∈
    {m.ownerClass.fields ∪ m.parameters ∪ inheritedFields(m.ownerClass)}) ∨
    (s.methodInvocation.invokeExpression is MethodInvocation methodInv2 ∧
    Field f = isGetter(methodInv2.methodDeclaration) ≠ null ∧
    f ∈ {m.ownerClass.fields ∪ inheritedFields(m.ownerClass)}) ∨
    s.methodInvocation.invokeExpression = null)
    return s.methodInvocation.declaringMethod) ∨

    (size of m.methodBody.statements = 1 ∧
    ∃ ReturnStatement r ∈ m.methodBody.statements where
    r.returnedExpression is MethodInvocation methodInv ∧
    methodInv.declaringClass.type ∈ program.classTypes ∧
    methodInv.declaringMethod.returnType = m.returnType ∧
    ((methodInv.invokeExpression is VariableAccess v ∧ v.declaration ∈
    {m.ownerClass.fields ∪ m.parameters ∪ inheritedFields(m.ownerClass)}) ∨
    (methodInv.invokeExpression is MethodInvocation methodInv2 ∧
    Field f = isGetter(methodInv2.methodDeclaration) ≠ null ∧
    f ∈ {m.ownerClass.fields ∪ inheritedFields(m.ownerClass)}) ∨
    methodInv.invokeExpression = null)
    return methodInv.declaringMethod)
```

```
(Method or null) $finalNonDelegateMethod$(Method $m$) ≡
    $nonDelegateMethod = m$
    while($delegatedMethod$ = isDelegate($nonDelegateMethod$) ≠ null)
        $nonDelegateMethod = delegatedMethod$
        finalNonDelegateMethod($nonDelegateMethod$)
    if $nonDelegateMethod = m$.declaringMethod
        return null
    else
        return $nonDelegateMethod$
```

Appendix B

```
boolean $modifiesDataStructureInTargetClass$(Method $m$, Class $t$) ≡
    for $i$ = 1 to size of $m$.parameters
        $parameter = m$.parameters[$i$]
        $F$ = one-to-manyAssociationRelationships($t$, $parameter$.type)
        if $F \neq \varnothing$
            for $j$ = 1 to size of $m$.methodBody.methodInvocations
                $methodInv = m$.methodBody.methodInvocations[$j$]
                if $methodInv$.declaringClass = $t \wedge$
                $pos$ = positionOfArgument($methodInv$, $parameter$) ≠ -1
                    $methodDecl = methodInv$.declaringMethod
                    $methodDeclParam = methodDecl$.parameters[$pos$]
                    for $k$ = 1 to size of $F$
                        Field $f = F[k]$
                        if modifiesDataStructure($methodDecl$, $f$, $methodDeclParam$)
                            return true

(set of Field) $one\text{-}to\text{-}manyAssociationRelationships$(Class $fromClass$, Type $toType$) ≡
    return $f \in fromClass$.fields
    where ($f$.type is ArrayType $aType \wedge aType$.type = $toType$) $\vee$
    elementTypeOfCollection($f$) = $toType$

boolean $modifiesDataStructure$(Method $m$, Field $f$, Parameter $param$) ≡
    if $f$.type is ArrayType $aType$ return
        $\exists$ $assignment \in m$.methodBody.assignments
        where $assignment$.leftHandSide is arrayAccess of VariableAccess $v_1 \wedge$
        $v_1$.declaration = $f \wedge aType$.type = $param$.type $\wedge$
        $assignment$.rightHandSide is VariableAcess $v_2 \wedge v_2$.declaration = $param$
    else if $elementType$ = elementTypeOfCollection($f$) ≠ null return
        $\exists$ $methodInv \in m$.methodBody.methodInvocations
        where $methodInv$.invokeExpression is VariableAccess $v_1 \wedge v_1$.declaration = $f \wedge$
        $methodInv$.name ∈ {add, remove, addElement, removeElement,
        set, setElementAt, insertElementAt, put} $\wedge$
        positionOfArgument($methodInv$, $param$) ≠ -1 $\wedge$ $elementType = param$.type
```

* arrayAccess of variable *array* is expression *array*[*indexExpression*]