TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ALGORITHMS

Page

LIST OF ABREVIATIONS

MDE	Model Driven Engineering
GP	Genetic Programming
GA	Genetic Algorithm
NSGA-II	Non-dominated Sorting Genetic Algorithm
IGA	Interactive Genetic Algorithm
CASCON	Conference of the Center for Advanced Studies on Collaborative Research
SQJ	Software Quality Journal
MOREX	MOdel REfactoring by eXample
JSEP	Journal of Software: Evolution and Process
MOREX+I	MOdel REfactoring by eXample plus Interaction
SSBSE	Symposium on Search Based Software Engineering
JASE	Journal of Automated Software Engineering
SA	Simulated Annealing
FD	Functional Decomposition

OO	Object-Oriented
NOM	Number Of Methods
JCC	Java Code Conventions
GQM	Goal Question Metric
VERSO	Visualization for Evaluation and Re -engineering of Object-Oriented SOftware
CBR	Check-Based Reading
PBR	Perspective-Based Reading
CSP's	Communicating Sequential Processes
CSP	Constraint Satisfaction Problem
C-SAW	Constraint-Specification Aspect Weaver
ECL	Embedded Constraint Language
OCL	Object Constraint Language
DL's	Description Logics
RCA	Relational Concept Analysis
EMF	Eclipse Modeling Framework

AGG	Attributed Graph Grammar
RCA	Relational Concept Analysis
HC	Hill Climbing
CBO	Coupling Between Classes
QVT	Query View Transformation
ATL	Atlas Transformation Language
SOR	Sequence Of Refactoring

INTRODUCTION

Research Context

Software systems are constantly evolving to cope with the changing and growing business needs. Software maintenance is the cornerstone of software evolution. Indeed, maintaining software and managing its evolution after delivery represents more than 80% of the total expenditure of the development cycle (Pressman, 2001). According to the ISO/IEC 14764 standard, the maintenance process includes the necessary tasks to modify existing software while preserving its integrity (ISO/IEC, 2006). One of these widely used techniques is software restructuring which is commonly called refactoring in object oriented systems.

According to Fowler (Fowler, 1999), refactoring is the process of improving the software structure while preserving its external behavior. Most of existing refactoring studies focus more on the code level. However, the rise of model-driven engineering (MDE) increased the interest and the needs for tools supporting refactoring at the model-level. Indeed, models are primary artifacts within the MDE approach and it has emerged as a promising approach to manage software systems' complexity and specify domain concepts effectively (Douglas, 2006). In MDE, abstract models are refined and successively transformed into more concrete models including executable source code. MDE activities reduce the development and maintenance effort by analyzing and mainly modifying systems at the model level instead of the code level. One of the main MDE activities is model maintenance (model refactoring) defined as different modifications made on a model in order to improve its quality, adding new functionalities, detecting bad designed fragments that corresponds to design defects, correcting them, modifying the model.

Model refactoring is a process that involves several activities including the activities of identifying refactoring opportunities in given software and determining which refactorings to apply. In this thesis, we are concerned with the two important problems: (1) detection of design defects in class diagrams and (2) correction of these design defects by suggesting

refactorings. In the next section, we will describe in details the challenges addressed by our proposal.

Problem statement

Despite the advances in design defects detection and model refactoring fields, we identify some problems related to the automation of these two processes.

When dealing with the automation of design defects detection, a number of challenges should be addressed:

1. Some detection approaches provided a way to guide the manual inspection of designs defects (e.g., (Tiberghien et al., 2007)). These approaches are not effective mostly because of the fact that manual inspection is time-consuming.
2. The majority of detection methods are based on designers' interpretations to detect the design defects. There is no consensual definition of the symptoms of the design defects, since interpretations might be different to describe this latter. The design defects that are commonly recognized in the literature such as the Blob (Fowler, 1999), deciding which classes are Blob candidates depends on the designer's interpretation. It is not trivial to define an appropriate threshold for the software metric (i.e., the size of a class) to consider a class A as blob. Class A could be interpreted as blob by a given designer community that could not be in another one.
3. The existing works defined detection rules based on a tedious domain analysis. Therefore, these rules are not scalable during the detection process. Thus, it is difficult to edit these detection rules even when the number of false positives is high.

Regarding the automation of model refactoring, most of the existing approaches for automating refactoring activities at the model level are based on declarative rules expressed as assertions or graph transformations or refactorings related to design patterns' applications. However, a complete specification of refactorings requires an important number of rules which represents a real challenge. When defining these rules we are still faced with this kind

of problems such as: (1) Incompleteness or missing rules; (2) Inconsistency or conflicting rules and (3) Redundancy or the existence of duplicated or derivable rules. Another common issue to most of these approaches is the problem of sequencing and composing refactoring rules. In addition, majority of these approaches offer semi-automatic tools because some key steps in the refactoring process requires the intervention of an expert to be accomplished.

Research motivation

The motivation of this research project is to help software designers to correct bad design practices in class diagram by automating the detection of design defects and the suggestion of the refactorings to correct these design defects.

Research objectives

The main goal of this thesis project is to propose an approach that supports software designers and developers during the model refactoring process and, in particular, the refactoring of UML class diagrams. To this end, we have identified the following specific objectives:

I. Detection of design defects in class diagrams. This includes:

   A. Designing techniques to identify defects in class diagrams, and
   B. Implementing and evaluating these techniques on existing class diagrams.

II. Correction of defects by suggesting refactorings. This includes:

   A. Designing techniques that generate correct sequences of refactorings to improve the quality of class diagrams.
   B. Implementing and evaluating these techniques on existing class diagrams.

Overview of the research methodology

To achieve our research objectives, we propose an approach for the detection and correction of design defects in class diagrams by applying heuristic search methods. To circumvent the issues mentioned in problem statement section, we consider the problem of detecting and correcting design defects, commonly known as refactoring, as a combinatorial optimization problem where a solution (i.e., a detected design defect or an appropriate sequence of refactorings) can be automatically generated from a limited number of examples of defects using heuristic searches (e.g., Genetic Programming (GP), Genetic Algorithm (GA), Interactive Genetic Algorithm (IGA) and Non-dominated Sorting Genetic Algorithm (NSGA-II)).

Figure 0.1 presents an overview of the research methodology to achieve the research objectives in three phases:

Phase 1: **Literature review**

Phase 1 of the research methodology consists of analyzing the literature related to the design defects detection, and the literature related to correction at the model level (model refactoring), identifying the weaknesses of the existing works on design defects detection and model refactoring.

Phase 2: **Design defect detection**

Phase 2 of the research methodology aims to design and implement techniques that detect design defects in class diagrams.

Phase 3: **Refactoring suggestion**

Phase 3 of the research methodology aims to design and implement techniques that suggest refactoring of class diagrams.

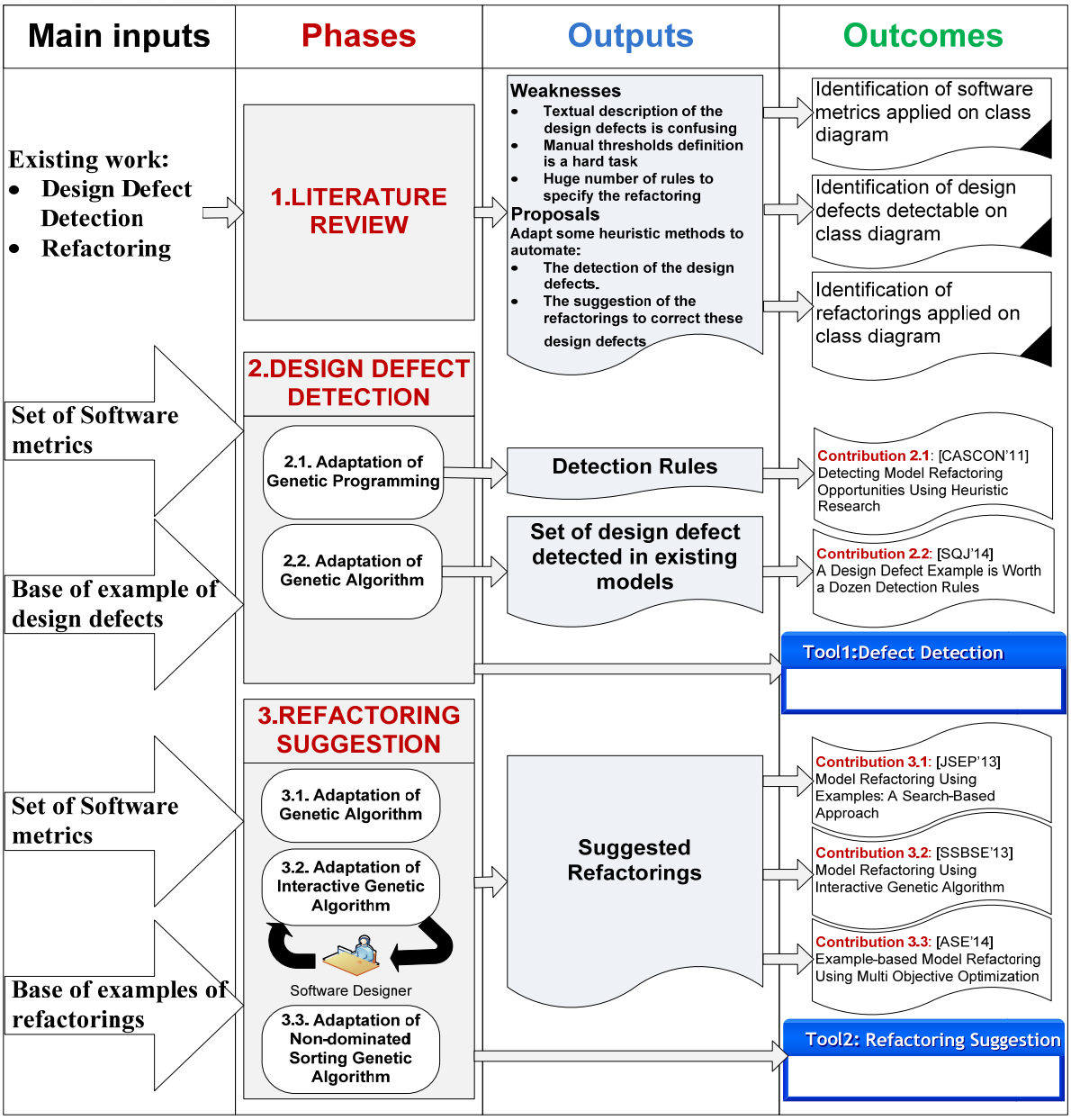


Figure 0.1 Overview of the research methodology

Detailed research methodology

Phase 1: **Literature review**

The objective of this phase is to understand the weaknesses and challenges of the existing works on design defects detection and model refactoring by analyzing the literature related to these two fields.

Phase 2: **Design defect detection**

The objective of this phase is to automate the design defects detection in class diagrams. This phase of the research methodology is separated into two major contributions:

Contribution 2.1: Adaptation of GP

In this contribution, we proposed an approach to generate detection rules from instances of design defects. This is achieved using GP which takes as inputs a base (i.e. a set) of defect examples and a set of software metrics. The rule generation process combines quality metrics (and their threshold values) within rule expressions. A tool was implemented to evaluate the approach. This approach was published in: Conference of the Center for Advanced Studies on Collaborative Research (CASCON) (Ghannem et al., 2011).

Contribution 2.2: Adaptation of GA

In this contribution, we proposed an approach that exploits examples of design defects and a heuristic search technique (i.e., GA) to automatically detect design defects on a given model and specifically in class diagrams. The approach takes as inputs a base (i.e. a set) of defect examples and a set of software metrics and it generates as outputs a set of design defects detected in the model under test. This approach was evaluated on four large open source systems, and a tool was implemented to this end. This approach was accepted in the Software Quality Journal (SQJ) (Ghannem et al., 2014a).

Phase 3: **Refactoring suggestion**

The objective of this phase is to automate the refactoring in class diagrams. This phase of the research methodology is separated into three major contributions:

Contribution 3.1: Adaptation of GA

In this contribution, we proposed MOREX (MOdel REfactoring by eXample), an approach to automate model refactoring using GA. MOREX relies on a set of refactoring examples to propose sequences of refactorings that can be applied on a given object-oriented model. The approach takes as input a set of examples of refactored models and a list of software metrics

and it generates as output a sequence of refactoring operations. We implemented a plug-in within the Eclipse™ development environment to support our approach. The plug-in supports many heuristic-based algorithms (GA, IGA, NSGA-II) for refactoring and hence enables to enter many controlling parameters depending on the chosen algorithm. The approach was published in Journal of Software: Evolution and Process (JSEP) (Ghannem et al., 2014c).

Contribution 3.2: Adaptation of IGA

In this contribution, we proposed MOREX+I: MOdel REfactoring by eXample plus Interaction) a model refactoring approach based on IGA. Two types of knowledge are considered in this approach. The first one comes from the examples of refactorings. The second type of knowledge comes from the designer's knowledge. The proposed approach relies on a set of refactoring examples, set of software metrics and designer's feedbacks to propose sequences of refactorings. The approach was published in the Symposium on Search Based Software engineering (SSBSE) (Ghannem et al., 2013).

Contribution 3.3: Adaptation of NSGA-II

In this contribution, we proposed a multi-objective optimization approach to find the best sequence of refactorings that maximizes both the structural and the semantic similarities between a given model (i.e., the model to be refactored) and a set of models in the base of examples (i.e., models that have undergone some refactorings). To this end, we adapted NSGA-II which aims at finding a set of representative Pareto optimal solutions in a single run. The approach takes as input a base of examples of models and their subsequent refactorings and a list of software metrics and it generates as output a set of optimal solutions of refactorings sequences. The approach was submitted to the Journal of Automated Software Engineering (Ghannem et al., 2014b).

Roadmap

The remainder of this dissertation is organized as follows:

Chapter 1 reviews related work on software refactoring; Chapter 2 reports our contribution for design defects rules generation published in CASCON (Ghannem et al., 2011). Chapter 3 presents our by-example approach to detect design defects accepted in SQJ (Ghannem et al., 2014a). Chapter 4 details our by-example model refactoring approach within mono-objective perspective published in JSEP (Ghannem et al., 2014c). Chapter 5 presents our model refactoring approach based on interactivity with designer published in the SSBSE (Ghannem et al., 2013). Chapter 6 details our model refactoring approach in multi objective perspective submitted to the JASE (Ghannem et al., 2014b). Finally, the conclusion of this dissertation and some directions for future research.

CHAPTER 1

LITERATURE REVIEW

The chapter presents a survey of existing works in two research areas: (1) detection of design defects and (2) correction of design defects (model refactoring) and identifies the limitations that are addressed by our contributions.

The structure of the chapter is as follows. Section 1.1 introduces some basic and relevant definitions. We survey exiting works on the detection of design defects at code and model levels in section 1.2. Section 1.3 discusses the state of the art in correcting design defects and especially model refactoring. Finally we summarize the limitations of reviewed works in section 1.4.

1.1 Basic concepts

In this section, we define the concept of design defect. Then, we introduce the notion of refactoring that aims to restructure a software system while preserving its behaviour by correcting its design defects.

1.1.1 Design defect

Design defects are common and recurring design problems that results from «bad» design choices (Brown et al., 1998). They affect the software development cycle especially the maintenance task by making it difficult to be accomplished. Unlike code defects (also called «code Smells» (Fowler, 1999)), which means the errors at the source code, design defects describe the defects that occur in the model level. In (Brown et al., 1998), the authors defined a taxonomy of design defects. In our thesis project, we will focus only on design defects that could affect the model such as:

Blob (called also Winnebago (Akroyd, 1996) or God class (Brown et al., 1998)): it is an object (class) with a lion's share of the responsibilities, while most other objects only hold data or execute simple processes. It is also called a «Controller class» which depends on data in their associated classes. It is a class with many attributes and methods and a weak cohesion. Figure 1.1 shows a blob example where the «Library_Main_Control» class contains a large number of methods and attributes.

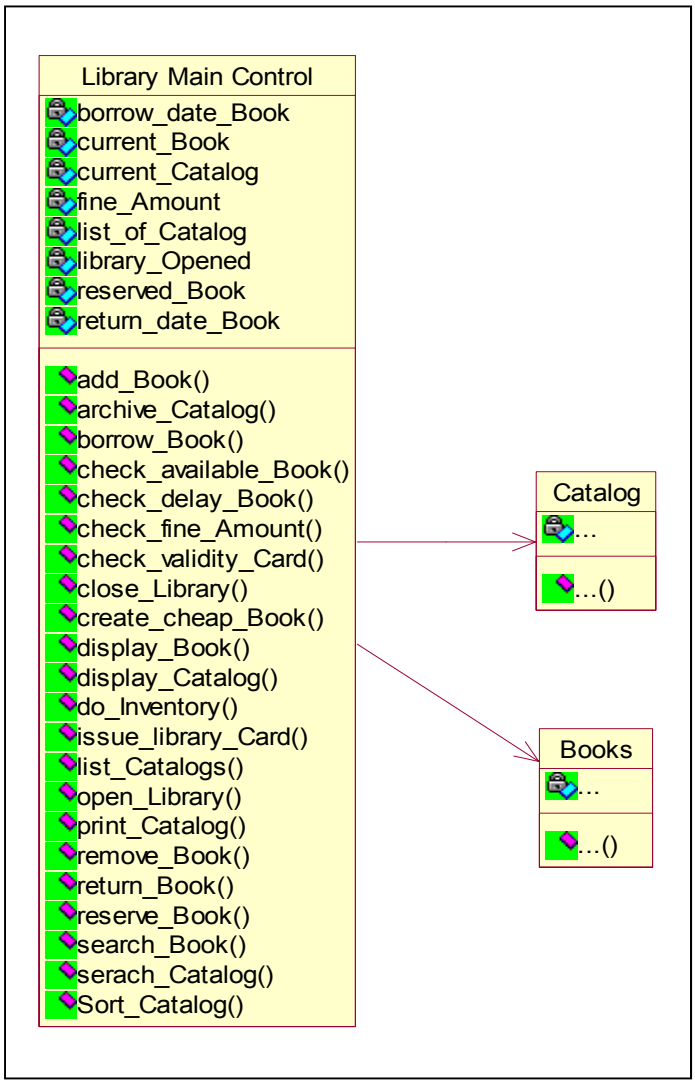


Figure 1.1 Example of Blob - Extracted from (Brown et al., 1998)

Functional decomposition: (Brown et al., 1998): it is an object (class) which name is usually a function name (e.g. Calculate_Payment_Schedule (see Figure 1.2)). All the attributes of this class are private and used only by this class. The inheritance and polymorphism which characterise the OO paradigm are hardly used. The class that shows this design defect is usually associated to classes which lead many attributes and implements few methods. Moreover, this class provides only one function.

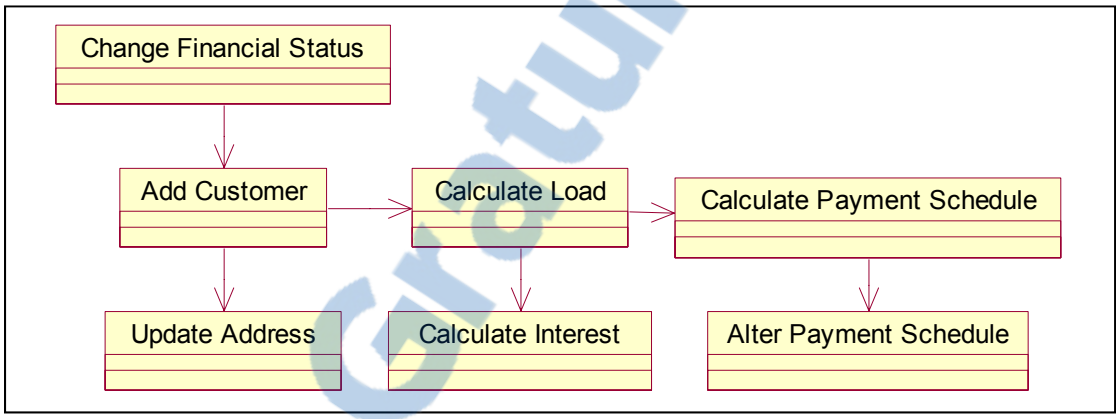


Figure 1.2 Example of Functional decomposition - Extracted from (Brown et al., 1998)

Data class: it is an object that encapsulates only data. The only methods that are defined by this class are the getters and the setters.

1.1.2 Refactoring

Refactoring is the action to restructure a software system in order to improve its quality while preserving the behaviour of the software application (Fowler, 1999). To our knowledge, the concept of refactoring was introduced for the first time by Opdyke (Opdyke, 1992) who gave a full description of some refactorings. Subsequently, Fowler and Beck (Fowler and Beck, 1999) provided more comprehensive view of the concept, and proposed 72 refactorings. These refactorings focus only on what they called «bad-smells» in the code. The 72 refactorings are illustrated with examples of java source code.

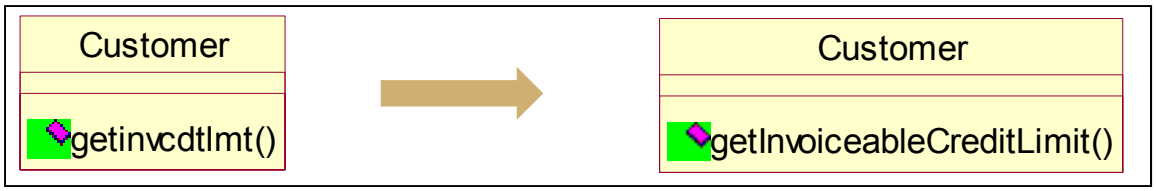


Figure 1.3 Rename_method - Extracted from (Fowler and Beck, 1999)

For example «Rename_method» refactoring aims to change the name of the method to better reveal its purpose (see Figure 1.3).

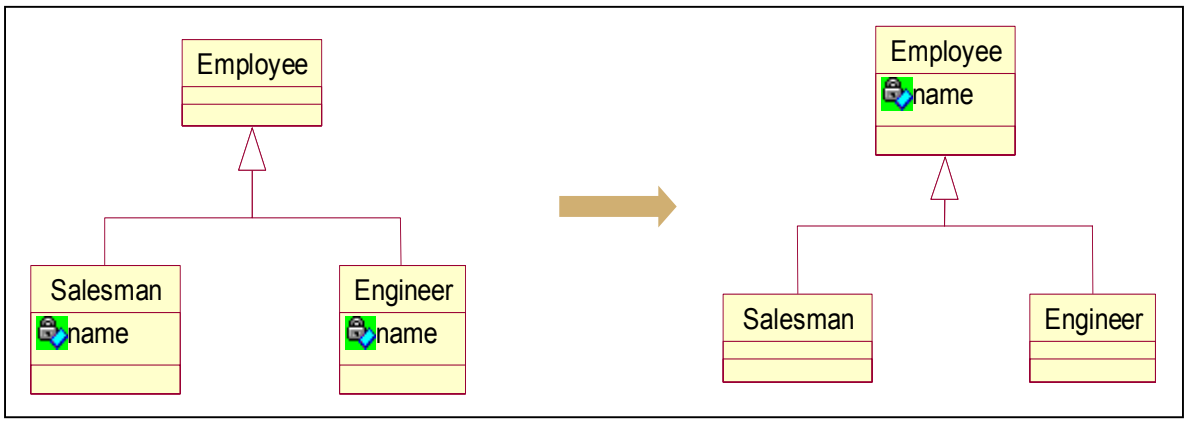


Figure 1.4 Pull-up-field - Extracted from (Fowler and Beck, 1999)

The «Pull-up-field» refactoring aims to move the same attribute from subclasses to the super-class as shown in Figure 1.4. We distinguish between two types of refactoring: Refactoring at source code defined in (Fowler and Beck, 1999) and refactoring in the model level which is very little studied. According to Munro (Munro, 2005), the refactoring is a process that involves first identifying where to apply refactoring, then choosing the appropriate solution (i.e. refactoring) and finally applying the refactoring.

For example, to refactor the blob class presented in Figure 1.1, we need to divide the «Library_Main_Control» class into several classes. To this end, we need to identify highly cohesive subsets of attributes and operations. Then, we move these subsets to other classes as illustrated by the Figure 1.5. To do that, we need to apply a couple of «move_method»,

«extract_class», «move_field» operations in order to obtain a class diagram without a blob class.

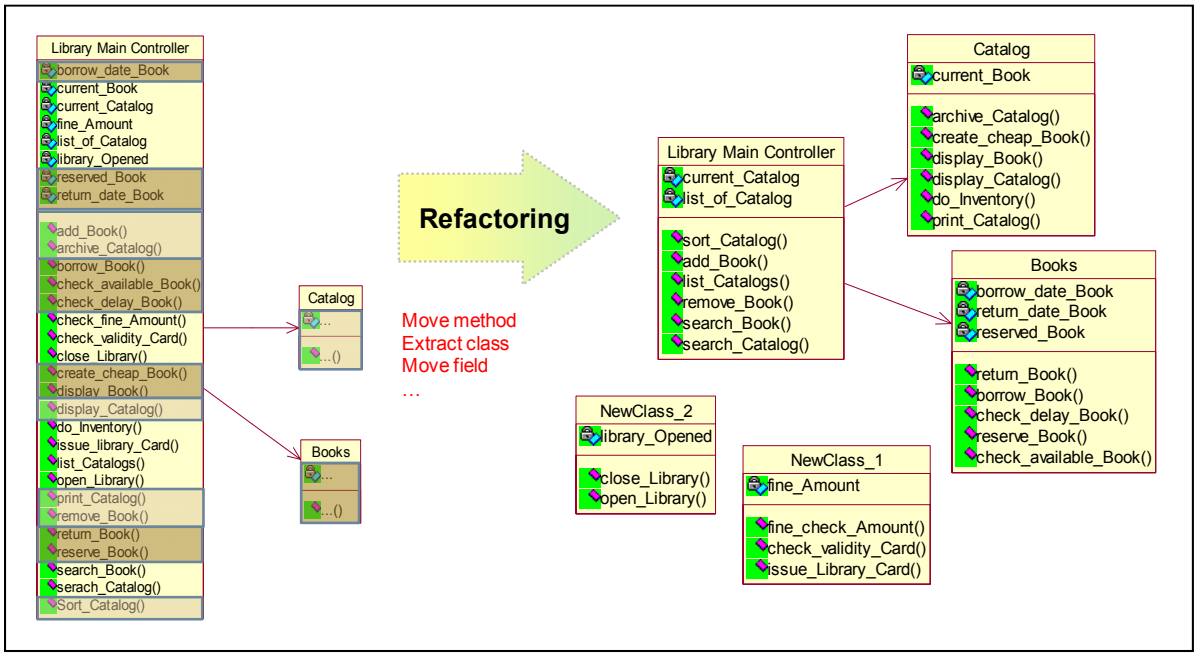


Figure 1.5 Refactoring the blob - Extracted from (Fowler and Beck, 1999)

1.2 Detection of defects

Many studies focus on the detection of design defects. We identify two levels that have been referred by the detection: source code and model. In this section, we report some relevant works in these two levels that have focused on defects in object-oriented as well as the techniques used for their detection systems.

1.2.1 Detection in source code level

One of the first works on defects in the code, is the work of Fowler et al. (Fowler, 1999). They defined a set of defects in the code called "Bad-Smells". They proposed a solution (or a series of refactoring) for each default. The authors believe that these defects in the code can be symptoms of the presence of design flaws.

In (Miceli et al., 1999), the authors used a set of object-oriented (OO) metrics to automate the detection of some symptomatic situations: a structure (e.g. class) in the source code or software design system which metrics values reveal weakness in the quality of the system. To do this, they proposed a model to estimate quality based on the correlation between the quality characteristics (e.g. maintainability) and metrics based on the opinions of experts. However, this process does not show what transformation should be applied to correct defects.

In (Tahvildar and Kontogiannis, 2004), the authors proposed a similar approach as (Miceli et al., 1999) by introducing the Framework QDR "Quality-Driven Reengineering", which uses the OO metrics as indicators to automatically detect a particular structure (class). Unlike (Miceli et al., 1999), the authors proposed the transformation that can improve the maintainability of OO system.

Another strategy based on OO metrics, was proposed by Marinescu (Marinescu, 2004) to locate design flaws in OO systems. The strategy relies on two mechanisms: filtering and composition. Filtering, also called data reduction (Hoel, 1954), aims to identify fragments with specific properties captured by a metric. The author defines two types of filters: 1) the marginal filter, which allows specifying an extreme limit for data, and 2) the interval filter which is a composition of two marginal filters. The selection of the filter to apply is based on the detection rules if they require thresholds or not. The composition is designed to correlate the sets of the obtained results by filtering to select symptoms that will provide a rule. For choosing thresholds, the author based on his expertise to increase the accuracy of the proposed strategy. The author defined four steps to build the strategy: 1) divide detection rule in a set of properties that can be captured by each one metric; 2) select the appropriate metrics for the properties; 3) find the right filter and finally 4) aggregate results using the tools of composition. This approach has been supported by a toolkit called PRODETECTION (Marinescu, 2004).

In (Munro, 2005), Munro addressed the issues of identification of "Bad-Smells" based on software metrics. Munro's work aims to automatically find opportunities of refactoring in software system (i.e., java source code). The author defined the identification processes for two "Bad-Smells («Lazy_class» and «Temporary_field») based on a combination of conventional metrics and new metrics. He starts by defining a bad smell description framework which has three main parts: 1) the name of the bad-smell in an informal description based on (Fowler, 1999); 2) the characteristics identified from the informal description and 3) design heuristics taken from the literature (Marinescu, 2004; Riel, 1996) relating to the characteristics defined in the second part. Thereafter, the author defines a detection rule for each "Bad-Smells" based on metrics within a framework which include the name, the software metrics potentially applicable to identify the defect and finally the detection algorithm illustrated in Figure 1.6.

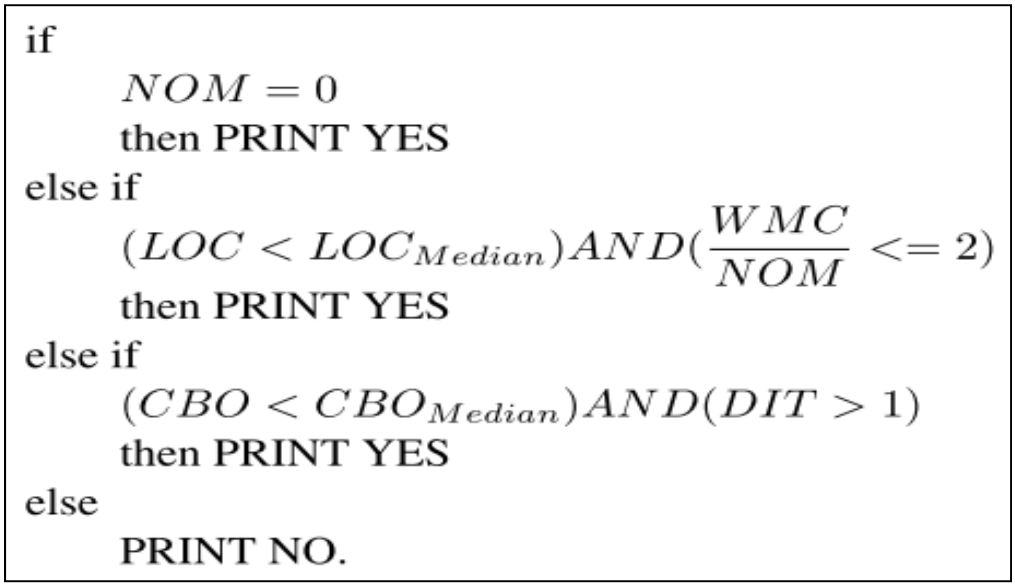


Figure 1.6 Algorithm to detect Lazy Class - Extracted from (Munro, 2005)

An algorithm is defined to detect one "Bad-Smell". It includes a set of rules as: **If** (condition) **then** Yes/No. Figure 1.6 shows an algorithm to detect the defect «Lazy_class». The condition contains a test on one or more software metrics calculated on the software system entity (e.g. Class). If the condition is true then it confirms the presence of this design defect otherwise there is no design defect. For example if the metric «Number_Of_Methods» (NOM) is equal

to zero then we can confirm the presence of the defect «Lazy_class». The author proposed a semi-automatic strategy based on an Integrated Development Environment Metrics (eclipse plug-in) and other adapted tools. For validating his strategy, the author applied it on a small system. The results showed only one false positive in the case of default «Lazy_ Class» and none in the case of «Temporary_field». This proposal represents a significant advance in the specification of defects based on rules. However, defects are textual descriptions and they can be interpreted in various ways (e.g., the same symptom could be associated to many design defects types). In addition, the author has not provided a way to automate its technical defect detection and did not explain his choice of thresholds.

Alikacem et al. (Alikacem and Sahraoui, 2006) proposed an approach based on the detection to evaluate the quality of OO systems. They proposed a quality rules description language based on metrics. For determining the metrics' thresholds, the authors used the fuzzy logic technique. This technique is applied in case metrics' thresholds are very difficult to setup: for exampole the rule: «methods should not be large». In this case, the threshold of the method's size is replaced by a range of values. Fuzzy logic variables may have a truth value that ranges in degree between 0 and 1 (e.g. fuzzy variable size - see Figure 1.7).

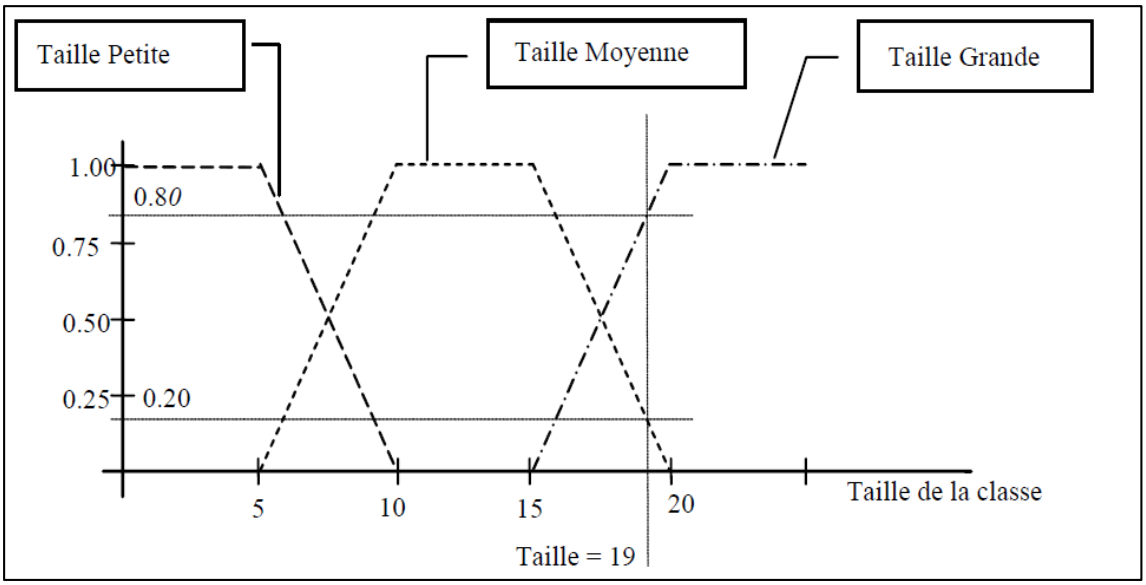


Figure 1.7 Fuzzy variable Size - Extracted from (Alikacem and Sahraoui, 2006)

The process of the detection approach (Alikacem and Sahraoui, 2006) is based on three steps:

1. Representation of source code: consists in extracting from the source code useful information for the evaluation of quality rules. Then, represent this information in a predefined meta-model in order to easily handle, on the one hand, and analyze regardless of programming language, on the other hand;
2. Collection of metrics: consists of choosing the important metrics that can help in the detection of defects based on literature and expert opinions by using the Goal Question Metric (GQM) approach.
3. Detection of nonconformities: consists in evaluating the description of quality rules via an interpreter. The evaluation is based on metrics and structural information contained in the representation of the source code.

The proposal of (Alikacem and Sahraoui, 2006) is based on three kinds of rules: 1) design heuristics (Marinescu, 2004; Riel, 1996); 2) anti-patterns (Brown et al., 1998) and 3) coding conventions defined in Java Code Conventions (JCC). A rule is represented by a set of conditions applied on metrics as illustrated in Figure 1.8. In this example, they detect the BLOB of a class (*c*) based on the metrics NOM (if it is large) and depth in the inheritance tree (DIT) (if it is low). Then, they calculate the same metrics on each class associated with *c* (if NOM is small and DIT is low) to see if they are «Data_classes». The approach was tested on only one type of defect which is the Blob.

IF *NOM(c) Grande* ***&&*** *DIT(c) Faible* ***THEN*** *Architecturel_BLOB(c) Grand*

Pour chaque classe a de l'ensemble de classes associées A(c)

IF *NOM(a) Petite* ***&&*** *DIT(a) Faible* ***THEN*** *Potentiel_classe_données(a) Grand*

IF *Architecture_BLOB(c) Grand* ***&&*** *Potentiel_classe_données(A(c)) Grand* ***THEN*** *Classe BLOB*

Figure 1.8 Algorithm to detect the Blob defect - Extracted from (Alikacem and Sahraoui, 2006)

In (Tiberghien et al., 2007), the authors proposed a model for each group of defects that explains the steps to manually detect this defect in the code (e.g. detect anti-patterns BLOB and SWISS ARMY KNIFE - see Figure 1.9. They are based on the classification of defects proposed in (Mantyla et al., 2003) to build groups. The proposed method is simple, well defined but time-consuming. For example, the authors spent twenty hours to detect defects on the GanttProject project.

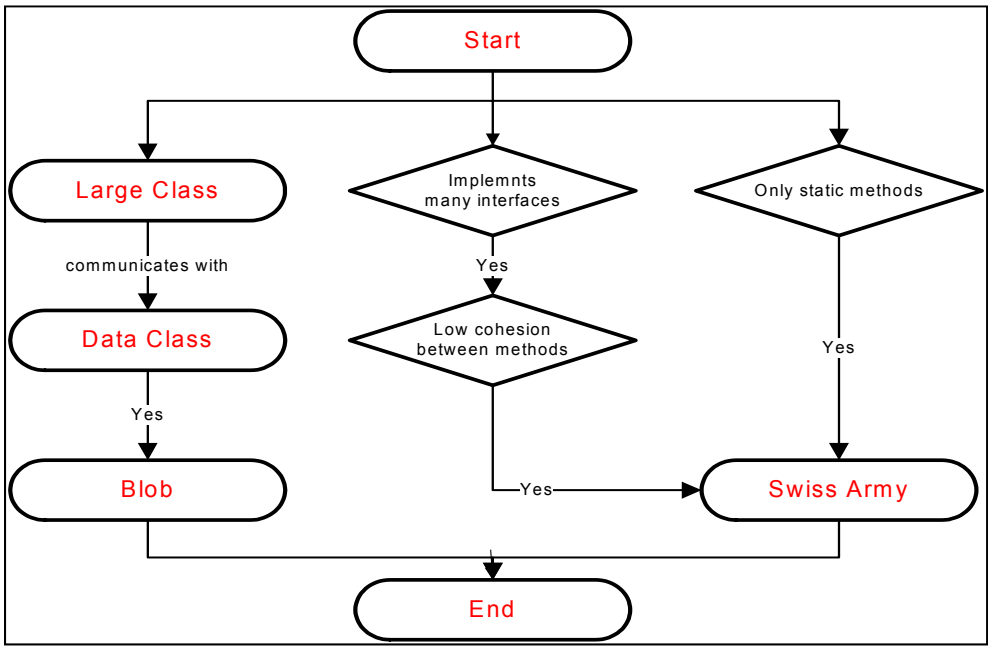


Figure 1.9 Diagram to detect Blob and Swiss Army knife anti-patterns - Extracted from (Tiberghien et al., 2007)

In (Moha et al., 2010), Moha et al. have proposed a technique called DETEX for defect detection. DETEX is an instance of a DEtection & CORection (DECOR): a method which includes all the necessary steps for defects detection and correction in both code and design levels. DETEX enables the automatic generation of detection algorithms based on rules. The process is organized as follows:

1. Define the rules to detect the occurrences of defects in the systems;
2. Generate automatically detection algorithms from those rules;

3. Apply these detection algorithms on model systems to obtain defects.

The classification and definition of defects is a manual process. The defects definition remains the step that requires more research efforts in the literature.

The majority of existing works on the defects detection at the source code handle a very large number of information (tens of thousands of lines of code). To address this problem, some works (Kothari et al., 2004) and (Dhambri et al., 2008) have used visualization techniques. In (Kothari et al., 2004), the authors presented a pattern-based framework for developing tool support to detect software anomalies represented in different colors. In (Dhambri et al., 2008), the authors presented a semi-automatic approach for visual detection of design anomalies. They adapted the Visualization for Evaluation and Re-engineering of object-oriented SOftware (VERSO) tool developed by (Langelier et al., 2005) to generate 3D representations of large software. These representations use quantitative (metrics) and structural (relations) data obtained by reverse engineering and extraction metrics tools (e.g. PADL (Gueheneuc and Amiot, 2004) and POM (Gueheneuc et al., 2004)). The authors have built a detection strategy for each design anomaly. They illustrated these examples of these strategies on few design anomalies (e.g. Blob see Figure 1.10).

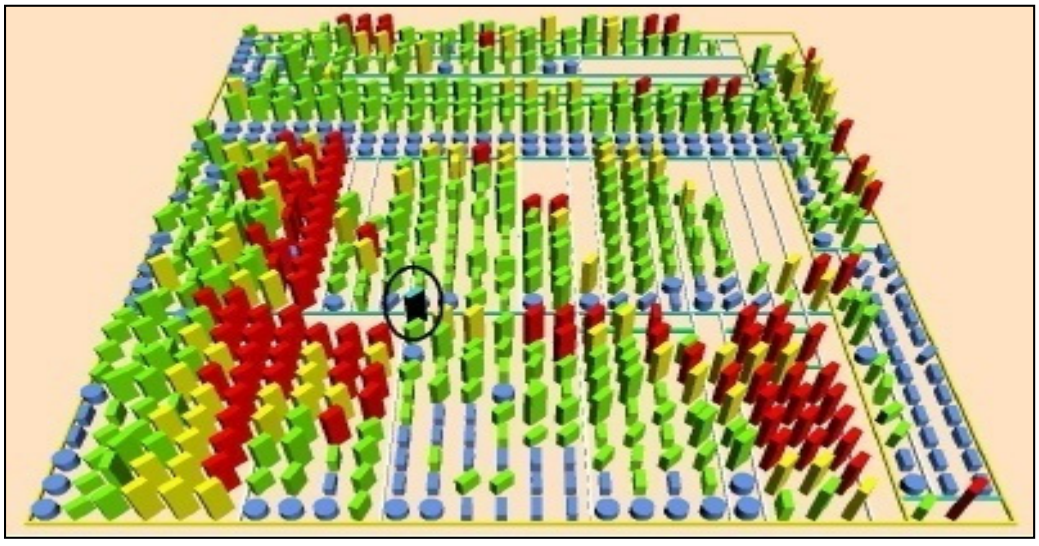


Figure 1.10 Example of blob detection - Extracted from (Dhambri et al., 2008)

The visualization metaphor presented in (Dhambri et al., 2008) was chosen specifically to reduce the complexity of dealing with a large amount of data. However, the visualization approaches are not readily applicable when evaluating large-scale systems. Moreover, the information visualized is for the most part metric-based, meaning that complex relationships can still be difficult to detect.

1.2.2 Detection in model level

Few studies have focused on the detection of design defects at the model level. The works cited above are intended to detect defects in models but not necessarily design defects but inconsistencies. We found that the majority of these works are more interested in inconsistencies in the model level. These approaches are based either on reading techniques or rules defined from the experience of some researchers. We can say that this area began (Brown et al., 1998) when they introduced for the first time the notion of anti-patterns that are considered design defects (e.g. BLOB, «Functional Decomposition», etc.). According to Brown et al, we cannot talk about anti-pattern without mentioning the design pattern (Gamma et al., 1995). As illustrated in Figure 1.11, the two concepts are related.

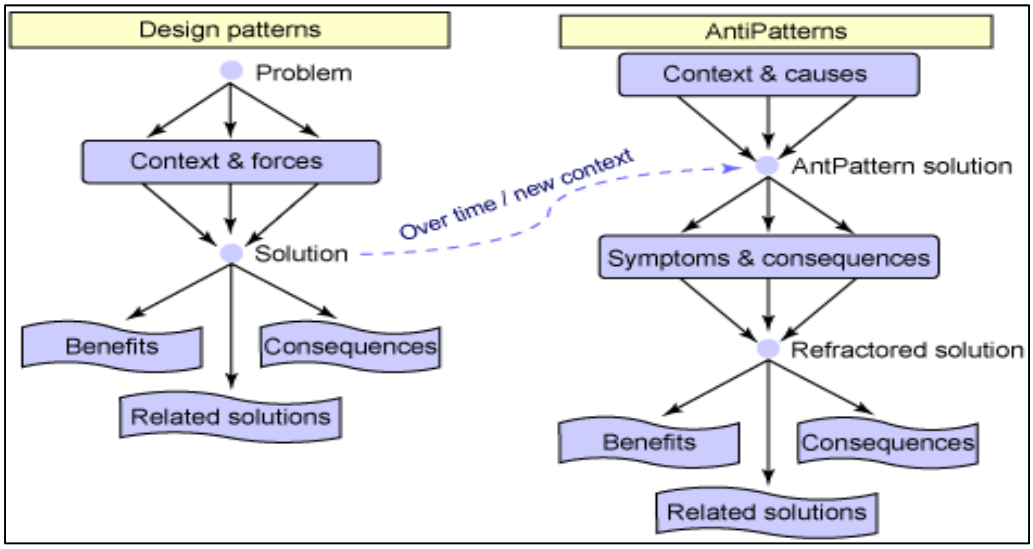


Figure 1.11 Relation between design patterns and anti-patterns - Extracted from (Brown et al., 1998)

The design patterns describe standard solutions to recurring problems in software design. Anti-patterns are common mistakes in software design. They appear during this phase either because of the absence or misuse of design pattern.

Travassos et al. (Travassos et al., 1999), proposed a well-defined process based on manual inspections and reading techniques to identify inconsistencies in some UML artifacts (class diagram, sequence diagram and state diagram). They used a taxonomy of defects defined in (Basili et al., 1996). An empirical study was conducted to evaluate the proposed approach. However, as the approach has not been automated, it remains difficult to apply on large systems.

In (Laitenberger et al., 2000), the authors presented a controlled experiment to compared two reading techniques («Check-Based Reading» (CBR) and «Perspective-Based Reading» (PBR)) used to detect inconsistencies in UML models (class diagram, collaboration diagram). According to the authors, the PBR is more efficient and less costly in terms of run time than CBR to detect inconsistencies in the mentioned UML diagrams. In this work, the types of defects found in the models are not explicitly identified.

In (Berenbach, 2004), the author has proposed what he called heuristics (rules of consistency in the case of (Bellur and Vallieswaran, 2006)) to create analytical models (use case diagram) and design model (class diagram) semantically correct. Each heuristic is described textually in both analysis model (e.g. model organization, use case definition, relationship between use cases and business model), and conceptual model. In this work, the author used a tool called «DesignAdvisor» developed by Siemens to implement these heuristics. He also used five models to validate the approach. The evaluation of these heuristics is not always automatic. The human intervention is essential to define the heuristics. In addition, these heuristics do not cover all the problems that can be found in an analysis and/or design model.

The study proposed by Leung et al. (Leung and Bolloju, 2005) aims to understand the defects frequently committed by novice analysts in the development of UML class diagrams. For this

purpose, they used a Framework proposed by Lindland et al. (Lindland et al., 1994) which allows to analyze the class diagrams quality in three categories: syntactic, semantic and pragmatic. These three categories address different aspects of quality that require increasingly more effort and expertise to achieve. For their study, the authors used 15 projects for beginners' analysts. They got 103 different types of defects in 15 projects, among which only 21 types of defects have been described. The proposed approach is manual and essentially based on the opinion of experts to detect and confirm the presence of defects. Defects designs are not part of the list of defects detected in this study.

Lange et al. (Lange and Chaudron, 2006) presented an empirical study on defects in UML models (use case, sequence and class diagrams). The study investigated the completeness of the software model by studying the completeness of these three diagrams and the consistency between them. To this end, the authors used eight defects (e.g. Message without Name (EnN), Message without Method (EcM), etc.). To assess the rate of design defects detection and the risk of misinterpretations caused by undetected defects, the authors used statistical techniques through a controlled experiment. The results' analysis showed that 96% of subjects detected the defect «class not instantiated in sequence diagram» (CnSD) while only 10% detected the defect called «multiple definitions of classes with equal names» (Cm) and most analysed types of defects are detected by less than 50% of subjects. To achieve this study, the authors proposed two types of defects classifications: 1) classification by detection rate (d-rate) and 2) classification by agreement measure (AgrM). According to the authors, d-rate and AgrM could be useful to prevent defects by providing guidelines in the creation of UML models and improve the correction of riskier defects. Although the empirical study presented in (Lange and Chaudron, 2006) is interesting, the authors did not address the design defect detection issue and focused only on inconsistencies. In addition, they did not show the impact of non-detected defects on model interpretation.

In (Bellur and Vallieswaran, 2006), the authors presented an approach to analyze consistencies in UML diagrams (use case, sequence, class, state transition, component, and deployment). They proposed a relational meta-model to represent different design entities

based on four different perspectives (requirements, deployment, source and development). Each perspective includes several diagrams. The analysis aims to study the impact of a change made in diagram on the other diagrams within the same perspective and between perspectives. To this end, they defined a set of consistency rules to build each UML diagram. Then, they evaluated these rules after applying changes based on a tool of checking consistency rules.

In (El Boussaidi et Mili, 2011), the authors proposed an approach to represent and implement design patterns based on the explicit representation of problems solved by design patterns. The design pattern is made up of triplet <MP, MS, T> where MP is a model of the problem solved by the pattern, MS is a model of the solution proposed by the pattern, and T is a model transformation of an instance of the problem into an instance of the solution. The approach consists of three main steps: 1) search for fragments that coincide with instances of MP in the input model; 2) marking the fragments, and 3) applying the transformation T to generate the MS. Marking (see Figure 1.12) is the process to recognize instances of model problems in the input model. According to the authors, a model of the problem (i.e., the representation of design defects) consists of a static model and temporal variation points «time hotspots». The recognition of the static part is seen as a graph pattern matching problem which was solved through constraint satisfaction techniques (CSP). The identification of "time hotspots" is done by comparing different versions of the class diagram of analyzed systems. Once this recognition was made, the model marking consists to assign tags to the elements of the class diagram (e.g. class, method, attribute, association, etc.) to indicate their role in the design problem that was detected. The focus on design patterns to solve problems, make the search space of the solution very limited. The process of detecting instances of problems which could be fixed by applying design patterns is semi-automatic process. The main problem with this approach is the characterization of design problems. Some problems can have an infinite number of representations.

The focus on design patterns to solve problems makes the search space of the solution very limited. El-Boussaidi and Mili proposed an approach that fully reasoned in the model level.

The process of detecting instances of problems which could be fixed by design patterns is a semi-automatic process. The main problem with this approach is the characterization of design problems. Some problems can have an infinite number of representations.

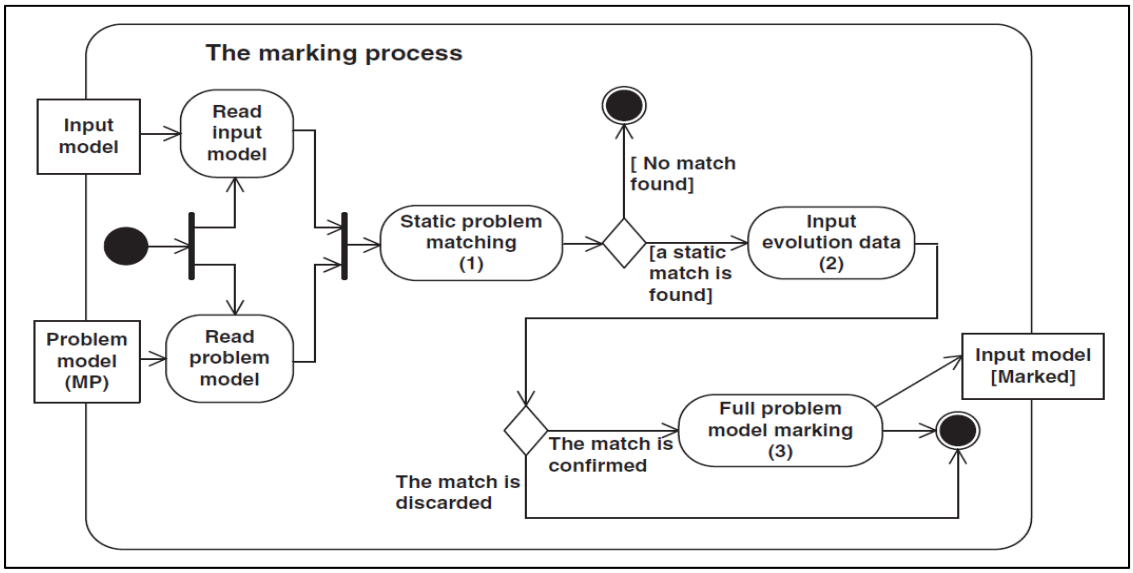


Figure 1.12 The process of detecting and marking instances of problem models - Extracted from (El-Boussaidi and Mili, 2011)

1.3 Synthesis on detection

We summarize in Table 1.1 all works presented in section 1.2.1 and 1.2.2 according to three criteria: detection support (automatic, semi-automatic or manual), the technique used and the abstraction level (model, source code).

All these works have contributed significantly to the detection of defects. However, most of these approaches used metrics to define detection rules. These rules are defined either manually or semi-automatically to identify the main symptoms that characterize a defect by combining metrics and/or structural information. For each defect, rules are expressed as a combination of metrics that requires significant effort calibration to find the right threshold values for each metric. The main limitation of these contributions is difficult to manually set up the threshold values for metrics in the detection rules. In addition, a comprehensive list of

design defects is necessary to specify all types of possible defects. On the other hand, the translation of the rules' symptoms is not obvious. Indeed, there is no consensus to define the symptoms for each design defect. If a consensus is found, the same symptoms may be associated with many types of defects which can compromise the precise identification of defect types. All these difficulties explain the low accuracy rate (number of true defects detected) and the high rate of false positives reported in the existing researches.

Regarding the approaches that tackled defect detection at the model-level, most of these works (Berenbach, 2004; Lange and Chaudron, 2006; Leung and Bolloju, 2005; Travassos et al., 1999), focused on the detection of model inconsistencies to improve the quality of the models. These inconsistencies are not real design defects. Their corrections do not mean that design defects were also corrected. The design defects and model inconsistencies are two different categories of defects and they require different types of model analysis.

Table 1.1 Synthesis on detection of defects

	Support			**Techniques**
	Automatic	*Semi-automatic*	*Manuel*	
Source code			(Fowler, 1999) (Tiberghien et al., 2007)	***Manual inspection***
	(Miceli et al., 1999) (Tahvildar and Kontogiannis, 2004)	(Marinescu, 2004) (Moha et al., 2010) (Munro, 2005)		***Textual Description, Detection rules***
		(Alikacem and Sahraoui, 2006)		***Fuzzy logic***
		(Kothari et al., 2004), (Dhambri et al., 2008)		***Visualization***
	(Sahin et al., 2014), (Kessentini et al., 2014)			***Heuristic search***
Model		(Lange and Chaudron, 2006)		***GQM***
			(Travassos et al., 1999), (Laitenberger et al., 2000),	***Reading techniques***

			(Leung and Bolloju, 2005)	
		(El-Boussaidi and Mili, 2011)		***CSP***
		(Berenbach, 2004)		***Heuristics***
			(Bellur and Vallieswaran, 2006)	***Manual Inspection***

1.4 Correction of design defects

Although, there is much research on refactoring code, refactoring models is still at a young stage of advancement. This is illustrated by the large number of work on defects in the code compared to those at the model level (see Table 1.1). To our knowledge, the first appearance of the concept of refactoring is attributed to (Opdyke, 1992). Based on some of the refactorings proposed by Opdyke (Opdyke, 1992), Fowler (Fowler, 1999) created a catalog of 72 refactorings targeting Java code. Later, Kerievsky (Kerievsky, 2004) proposed a catalog of refactorings that can restructure code by introducing design patterns.

Mens et al. (Mens and Tourwé, 2004) have specified five groupings that characterize the software refactoring area:

1. Refactoring activities: consists of identifying fragments of the model to refactor; determine what types of refactorings that should be applied to these fragments; ensure that once applied, refactoring preserves behavior models; automate the process of refactoring; assess the impact of refactoring on software quality criteria (complexity, legibility, adaptability) or process (productivity, cost, effort); and finally synchronize the refactored model and other artifacts such as source code, documentation, specifications, testing, etc.
2. Techniques and formalisms: aims to support the activities above (e.g. technique based on assertions, technique based on graphs and search based technique).
3. Artifacts to be refactored: for example source code or design (e.g. class diagram, state diagram, activities diagram).

4. Tools: according to the authors, tools must take into account usability, automation, reliability, configurability, scalability and coverage.
5. Process: describes how to integrate refactorings in: 1) software engineering process (e.g. legacy systems); 2) Agile software development process and 3) the software development based on frameworks and product lines.

In the following subsections, we review existing work on refactoring. We classify existing approaches into two categories: 1) traditional approaches to refactoring: these approaches rely on metrics and rules to specify refactorings; and 2) approaches that considered the refactoring as an optimization problem: these approaches rely mostly on search-based techniques.

1.4.1 Traditional approaches to software refactoring

In (Van Kempen et al., 2005), the authors proposed an approach to detect refactoring opportunities after detecting design defects (e.g. Controller Class) in UML diagrams (e.g. class diagram, state diagram) based on software metrics. The proposed approach consists of analysing theses metrics on a specific model (i.e. class diagram of SAAT «*Software Architecture Analysis Tool*»), then suggesting the refactorings that remedy to design defects already identified. In order to preserve the behaviour after applying refactorings, the authors considered the state diagram for each class to compare their traces before and after the refactoring. To this end, the authors used a formalism based on CSP «Communicating Sequential Processes». The approach was supported and validated by SAAT used to calculate metrics on UML models. The approach presented in (Van Kempen et al., 2005) considered only one type of refactoring to restructure the model to correct the design defect (i.e. class controller) identified in Saat class (see Figure 1.13 and Figure 1.14). The identification of the design defect and the verification of the preservation of the behaviour are manually defined in (Van Kempen et al., 2005) which has an impact on the effectiveness of the approach especially when applied to large systems.

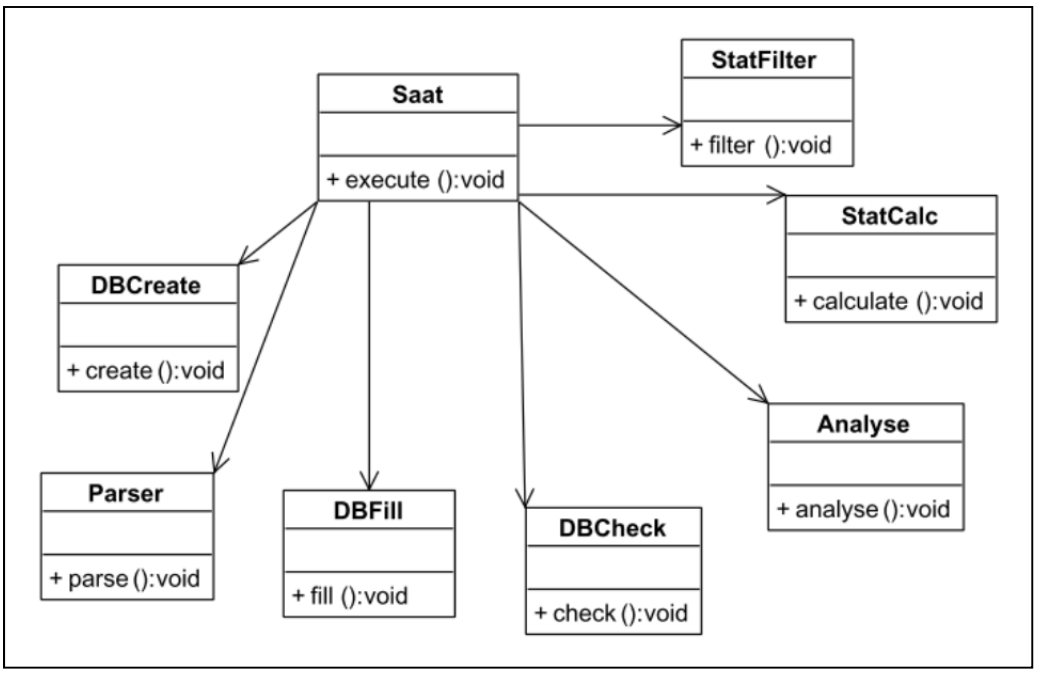


Figure 1.13 Class diagram of SAAT before refactoring - Extracted from (Van Kempen et al., 2005)

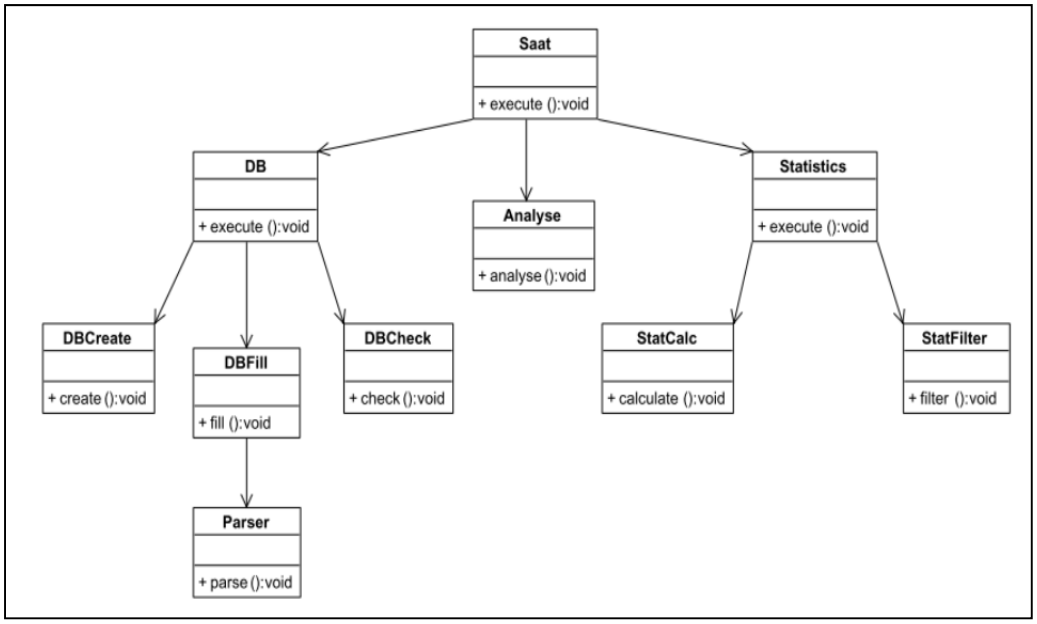


Figure 1.14 Class diagram of SAAT after refactoring - Extracted from (Van Kempen et al., 2005)

In (Gheyi et al., 2007) suggest to formalize a static semantics in Alloy to specify and prove semantics-preserving properties of model refactorings. Pretschner and Prenninger (Pretschner and Prenninger, 2007) proposed an approach to refactor state machines diagram based on logical predicates and tables. In (Ragnhild and D'Hondt, 2006), the authors proposed an approach that supports model refactoring based on forward-chaining logic reasoning engine. The automation of these approaches remains a challenge because the user intervention, in both description and selection of the type of correction, is necessary.

In (Zhang et al., 2005), the authors proposed a model refactoring strategy based on model transformation using C-SAW «Constraint-Specification Aspect Weaver» engine. The choice of this engine is due to its ability in exploring several scenarios through modeling aspects and performing updates (insertion, deletion) in the model. The authors introduced a specific pattern for refactoring (name of the refactoring, parameters, preconditions and strategies). They used examples of refactorings to validate their approach on generic models (e.g. refactoring «Extract Super-Class» (see Figure 1.15). The refactoring is specified in a formal way through the ECL «Embedded Constraint Language» language (i.e., an extension of OCL «Object Constraint Language»). The most tedious part of this approach is the manual specification of refactorings by domain experts.

Name: Extract Superclass
Parameters: selectedClasses, className
Preconditions:
1. The className for the new super class must be unique, i.e., no other classes have the same name.
2. All of the selected classes must have at least one common attribute.

Strategies:
1. Create a new superclass named as className.
2. Insert the common attributes into this superclass.
3. Delete the common attributes in each selected class.
4. Make an inheritance relationship from the superclass to the selected classes.

Figure 1.15 Refactoring pattern for «Extract SuperClass» - Extracted from (Zhang et al., 2005)

Biermann et al. (Biermann, 2010) and Mens et al. (Mens et al., 2007b) have used graph transformation theory as their foundation to specify model refactoring and relied on the formal properties to these refactorings. In (Biermann, 2010), the authors represent refactoring as a transformation rule (LHS «Left Hand Side» and RHS «Right Hand Side») where LHS represent the fragment of the model to refactor and RHS represent the fragment after refactoring. They used the EMF framework «Eclipse Modeling Framework» through its meta-model to define these transformation rules for six refactoring actions (CreateSuperclass, ConnectSuperclass, CheckAttribute, PullUpAttribute, DeleteAttribute, DeleteAnnotation). To test the consistency of such transformations, the authors used the Attributed Graph Grammar (AGG) system. Thus, the rules will be translated and inferred in the above tool for analysis. In (Mens et al., 2007b), the authors have proposed to represent refactoring as graph to be merged within the graph representing the model at hand. They have used critical pair analysis technique to detect structural merge conflicts. Many issues should be addressed in this proposal such as the completeness of the refactoring specification and the transformation rules (some refactoring require more than one rule, and sometimes even an infinite number of rules).

In the same direction, El Boussaidi and Mili (El-Boussaidi and Mili, 2011) presented an approach to apply changes based on graphs to generate a model solution (MS). To do this, refactoring rules are used to specify design patterns' applications. In this context, design problems solved by these patterns are represented using models and the refactoring rules transform these models according to the solutions proposed by the patterns. However, not all design problems are representable using models; i.e., for some patterns (e.g., Observer), the problem space is quite large and the problem cannot be captured in a single, or a handful problem models.

In (Moha et al., 2008b), the authors proposed an approach to automate correction of design defects based on Relational Concept Analysis (RCA). To this end, they detect all suspicious classes based on their previous proposal (Moha et al., 2006). For each suspicious class, they extract relational context family to be introduced into an RCA engine that generates the

corresponding concept lattices. Then, they interpreted these concepts and apply the appropriate refactorings. One design defect was considered in this work which is the blob. To correct this defect, the authors focused on cohesion and coupling information which affect the results that show a large number of false positives. A trade-off between cohesion and coupling remains a challenge due to the problem of subjectivity. Within an MDE context, (Moha et al., 2009) and (Reimann et al., 2010) have tried to propose a generic refactoring to foster their reuse based on roles model and kermeta, respectively. The limitation of these approaches is that not all refactorings could be instantiated from generic ones and reusing refactorings is not beneficial in all cases.

1.4.2 Search-based software refactoring approaches

More recently, several research studies have considered the refactoring as an optimization problem that can be solved using search-based techniques (e.g. (Seng et al., 2006), (Harman and Tratt, 2007), (O'Keeffe, 2008), (Jensen and Cheng, 2010), (Ben Fadhel et al., 2012) and (Kessentini et al., 2012)).

For example, a heuristic-based approach is presented in (Harman and Tratt, 2007; O'Keeffe, 2008) in which various software metrics are used as indicators for the need of certain refactorings. In (Harman and Tratt, 2007), the authors presented the refactoring as a heuristic search problem and used the concept of Pareto optimality (Pareto front) to improve this search. This front is used to exploit all possible optimal solutions (good quality). To this end, they adapted Hill Climbing (HC) algorithm and defined an objective function which consist to minimize the CBO «Coupling Between Classes» metric. The idea consists of calculating the CBO value on a system at hand, then applying the refactoring (i.e., move method) on a chosen method at random, and observing the behaviour of the CBO metric value. If the value increases, the applied refactoring operation is discarded and another one is tried. Otherwise they keep the refactoring in the sequence of refactorings in construction. Based on this process, the authors noticed a birth of many blobs. To remedy this problem, they have introduced a second metric called «Standard Deviation of methods Per Class» (SDMPC).

The result of this approach is a Pareto front that includes all possible optimal solutions. In (Seng et al., 2006), a genetic algorithm is used to suggest refactorings to improve the class structure of a system. The algorithm uses a fitness function that relies on a set of existing OO metrics. Both the approaches in (Seng et al., 2006) and (Harman and Tratt, 2007) were limited to the «Move Method» refactoring operation.

In (O'Keeffe, 2008), the authors present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics. The results of the experiments on five open-source systems showed that hill-climbing performs better than the other algorithms. In (Jensen and Cheng, 2010), the authors proposed an automated refactoring approach that uses genetic programming (GP) to support the composition of refactorings that introduce design patterns. The fitness function used to evaluate the applied refactorings relies on the same set of metrics as in (O'Keeffe, 2008) and a bonus value given for the presence of design patterns in the refactored design.

In (Kessentini et al., 2012), the authors proposed a by-example approach based on search-based techniques for model transformation. A Particle Swarm Optimization (PSO) algorithm is used to find the best subset of transformation fragments in the base of examples, that can be used to transform a source model (i.e., Class Diagram) to a target model (i.e., Relational Schema). Furthermore, the fitness function proposed in (Kessentini et al., 2012) relies on the adequate mapping of the selected transformation examples with the constructs of the model (e.g., class, relationship) to be transformed while our fitness function exploits the structural similarity between classes. This approach targets exogenous transformations (i.e., different source and target languages) and do not focus on the endogenous transformations that aim at correcting design defects.

In (Ben Fadhel et al., 2012), the authors proposed an approach that extracts refactoring operations from changes between two versions of the same model. They adapted the algorithm of genetic programming (GP) to detect the sequence of refactorings. Both the

detection and correction step are considered in this proposal but its output could be used as base of examples for a by-example approach to correct defects based on an heuristic method.

1.5 Synthesis on correction

According to our analysis of relevant works in model refactoring, we note that each considered research work has focus on a very specific aspect (activity) of the refactoring process. Mens et al. (Mens and Tourwé, 2004) defined three main activities for this process: 1) suggesting; 2) applying and 3) preserving behaviour. We considered these activities as criteria to classify these research works (see Table 1.2). We also enriched the classification by the variety of techniques and formalisms used to support these activities.

The first criterion is the automation of suggestion of refactoring supported by heuristic methods such as GP, HC, SA (e.g., (Harman and Tratt, 2007), (O'Keeffe, 2008) and (Ben Fadhel et al., 2012)) or RCA in (Moha et al., 2008b) and graph-based approaches (e.g., (Biermann, 2010; Mens et al., 2007b) and (El-Boussaidi and Mili, 2011)). The second criterion is the automatic application of refactoring on the relevant part of the model. According to (Mens et al., 2010), there are three types of model transformation: 1) the logic transformation based on the knowledge of various elements of the model, usually represented in a logical language such as Prolog or SQL; 2) the transformation based on graphs where the refactoring model corresponds to a graph rule which consists of a left part (precondition: design defect) and right part (post-condition: solution to the defect) to ensure the preservation of behavior; and 3) transformation based on imperative languages (e.g. Kermeta (Moha et al., 2009), «Query View Transformation» (QVT) and «Atlas Transformation Language» (ATL)). Finally, the preservation of the behavior after applying a refactoring is generally supported by formal techniques such as DL and CSP ((Van Kempen et al., 2005), (Ragnhild and D'Hondt, 2006)).

For the approaches which focused on preserving behaviour (e.g. (Van Kempen et al., 2005), (Ragnhild and D'Hondt, 2006)), most of them assumed that the detection and correction steps

were performed successfully. However, refactoring is wider than the preservation of the behavior which remains a crucial step to improve the quality of the models after applying the refactoring.

The approaches which considered the refactoring as a graph transformation, focused on some specific entities in the model and do not consider the complete model (e.g. (Zhang et al., 2005)). We also noticed that the detection step is implicit in these approaches, i.e., there is no explicit definition of the notion of design defect. Although these approaches offer visual support that makes them attractive to designers, they still rely on tools that are not easy to use (e.g., Constraint Specification Aspect Weaver engine). This makes it difficult for domain experts to specify the refactoring rules. Generally, most of the traditional approaches to refactoring are based on rules that are expressed either as assertions or graphic transformation. However, a complete specification of a refactoring requires, most of the time, a large number of rules. In addition, applying automatically these rules raises many issues related to possible conflicts and dependencies between suggested refactorings. Indeed, finding the appropriate and optimal sequence (order) of refactorings remains a challenging research problem.

Finally, as presented in the previous subsection, some approaches considered the refactoring problem as an optimization problem. Most of these approaches (e.g., (Seng et al., 2006), (Harman and Tratt, 2007), (Jensen and Cheng, 2010) and (Ben Fadhel et al., 2012)) suggest refactorings using a fitness function defined as a combination of metrics (i.e., 12 metrics in case of (Jensen and Cheng, 2010; O'Keeffe, 2008) and two metrics in the case of (Harman and Tratt, 2007)). However, improving the quality of software system in terms of metrics' values does not necessarily mean that the suggested refactoring will make sense. In addition, many of these approaches do not clearly identify the link between the defects and the suggested corrections.

Table 1.2 Synthesis on model refactoring

Approaches	Refactoring activities			Techniques
	Suggesting	Applying	Preserving behaviour	
(Van Kempen et al., 2005), (Van Der Straeten R. and D'Hondt, 2006), (Gheyi et al., 2007) and (Pretschner and Prenninger, 2007)			√	CSP, DL's and Forward Chaining Logic
(Zhang et al., 2005)		√		C-SAW
(Mens et al., 2007b), (Biermann, 2010)		√		Graphs
(O'Keeffe. and Cinneide, 2008), (Harman and Tratt, 2007), (Ben Fadhel et al., 2012), (Seng et al., 2006), (Jensen and Cheng, 2010) and (Kessentini et al., 2012)	√			Heuristic methods (PG, HC, SA)
(El-Boussaidi and Mili, 2011)	√	√		CSP, Graphs
(Moha et al., 2008a)	√	√		RCA
(Moha et al., 2009), (Reimann et al., 2010)		√		Kermeta, Role models

1.6 Limitations of existing works

We summarize in this section the limitations identified in the related works in both detection and correction areas:

In the detection area, we noticed that most of existing work have followed the pattern that consists of defining the design defect, identifying the symptoms of this design defect and finally, defining the detection rule to detect this design defect. However, some difficulties arise such as:

1. The difficulty to automate symptom's evaluation: it's hard to define the threshold values for each metric.
2. The difficulty to derive consensual rules to detect the design defects: there is no consensus to define symptoms of each design defect because of the diverging expert's

opinions. Sometimes, the same symptom could be associated to many design defect types.

In the correction area, we noticed the absence of the detection step in most of refactoring approaches and the semantic aspect in majority of search based approaches to suggest refactorings. However, it’s difficult to propose a refactoring solution for each design defect because of the non consensus to order refactorings to fix one problem. As consequence, it‘s difficult to automate the refactoring process.

CHAPTER 2

DETECTING MODEL REFACTORING OPPORTUNITIES USING HEURISTIC SEARCH

Adnane Ghannem[1], Marouane Kessentini[2] and Ghizlane El Boussaidi[1]

[1]Department of Software and IT Engineering, École de Technologie Supérieure,

1100 Notre-Dame West, Montreal, Quebec, (H3C 1K3) Canada

[2]Department of Computer and Information Science, University of Michigan - Dearborn

4901 Evergreen Road, Dearborn, MI 48128 USA

This paper has been published in CASCON [1]

ABSTRACT

Model-driven engineering (MDE) is an approach to software development where the primary focus is on models. To improve their quality, models continually evolve due, for example, to the detection of "bad design practices", called design defects. Presence of these defects in a model suggests refactoring opportunities. Most of the research work that tackle the problem of detecting and correcting defects, concentrate on source code. However, detecting defects at the model level and during the design process can be of great value to designers in particular within an MDE process. In this paper, we propose an automated approach to detect model refactoring opportunities related to various types of design defects. Using Genetic Programming, our approach allows automatic generation of rules to detect defects, thus relieving the designer from a fastidious manual rule definition task. We evaluate our approach by finding three potential design defect types in two large class diagrams. For all these models, we succeed in detecting the majority of expected defects.

---

[1] *Conference of the Center for Advanced Studies on Collaborative Research*

2.1 Introduction

Model Driven Engineering (MDE) is an approach to software development by which software is specified, designed, implemented and deployed through a series of models (Bull, 2008). Hence building appropriate models, evolving them and maintaining their quality are key activities when implementing an MDE approach.

Model maintenance is defined as different modifications made on a model in order to improve its quality, adding new functionalities, etc (Brown et al., 1998). This effort needs a lot of time and money from the total project cost. Thus, it is really important to propose automated solutions to improve model quality.

Different automated maintenance solutions were proposed in the literature (Kessentini et al., 2010; Khomh et al., 2009; Liu et al., 2009; Marinescu, 2004; Moha et al., 2010). The majority of these works are concerned with the detection and correction of bad design fragments, called design defects or refactoring opportunities (Fowler and Beck, 1999). Such defects include for example large classes in UML, long parameter list, etc. Detecting and fixing design defects is, to some extent, a difficult, time-consuming, and manual process (Fowler and Beck, 1999).

To insure detection of design defects, several approaches have been already proposed (Khomh et al., 2009; Liu et al., 2009; Marinescu, 2004). The large portions of these studies are based on declarative rule definition. These rules are manually defined to identify the symptoms that characterize a defect. These symptoms are described using metrics, structural, and/or lexical information. For example, large classes have different symptoms like the high number of attributes, relations and methods that can be expressed using quantitative metrics. However, in an exhaustive scenario, the number of possible defects to be manually characterized with rules can be very large. For each defect, rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric, threshold above which a defect is said to be detected.

Besides, one can notice the availability of defect repositories in many companies, where defects in projects under development are manually identified, corrected and documented. Despite its availability, this valuable knowledge is not used to mine regularities about defect manifestations. These regularities could be exploited both to detect defects, and to correct them.

Starting from this observation, we propose, in this paper, an approach to overcome some of the above mentioned limitations. Our approach is based on the use of defect examples generally available in defect repositories of software developing companies. In fact, we translate regularities that can be found in such defect examples into detection rule solutions. Instead of specifying rules manually for detecting each defect type, or semi-automatically using defect definitions, we extract these rules from instances of design defects. This is achieved using Genetic Programming (GP). Such proposal is very beneficial because: it does not require to define the different defect types, but only to have some defect examples; it does not require an expert to write rules manually; it does not require to specify the metrics to use or their related threshold values.

The remainder of this paper develops our proposals and details how they are achieved. Therefore, the paper is structured as follows. Section 2.2 is dedicated to the basic concepts related to our approach. In section 2.3, we give the motivations of our proposal. Then, section 2.4 details our adaptations of Genetic Programming to the model-defect detection problem. Section 2.5 presents and discusses the validation results. The related work in defect detection and correction is outlined in section 2.6. We conclude and suggest future research directions in section 2.7.

2.2 Basic concepts

To better understand our contribution, it is important to clearly define some relevant concepts to our proposal, including design defects and software metrics.

2.2.1 Design defects

We focus in this paper on the detection of a specific type of refactoring opportunities to improve model quality: design defects. Design defects, also called design anomalies, refer to design situations that adversely affect the development of models (Brown et al., 1998). Different types of defects, presenting a variety of symptoms, have been studied in the intent of facilitating their detection and suggesting improvement solutions.

In (Fowler and Beck, 1999), they define 22 sets of symptoms of common defects. These include large classes, feature envy, long parameter lists, and lazy classes. Each defect type is accompanied by refactoring suggestions to remove it. Brown et al. (Brown et al., 1998) define another category of design defects that are documented in the literature, and named anti-patterns.

In our approach, we focus on the detection of some defects that can appear in the model level and especially in class diagram. We choose from (Fowler and Beck, 1999) three important defects that can be detected in the model level: 1) Blob which is found in designs where one large class monopolizes the behavior of a system (or part of it), and other classes primarily encapsulate data. 2) Functional decomposition: it occurs when a class is designed with the intent of performing a single function. This is found in model (class diagram) produced by non-experienced object-oriented developers. 3) Poor usage of abstract class: it is happen when abstract classes are not used widely in the application design.

2.2.2 Quality metrics

Quality metrics provide useful information that help assessing the level of conformance of a software system to a desired quality such as *evolvability* and *reusability*. Metrics can also help detecting some design defects in software systems. The most widely used metrics are the ones defined by Genero et al. (Genero et al., 2002). For our defect detection process, we

select from this list of metrics only those that can be calculated on models (class diagram). These metrics include:

1. Number of associations (Naccoc): the total number of associations.
2. Number of aggregations (Nagg): the total number of aggregation relationships.
3. Number of dependencies (Ndep): the total number of dependency relationships.
4. Number of generalizations (Ngen): the total number of generalisation relationships (each parent-child pair in a generalization relationship).
5. Number of aggregations hierarchies (NAggH): the total number of aggregation hierarchies.
6. Number of generalization hierarchies (NGenH): the total number of generalisation hierarchies.
7. Maximum DIT (MaxDIT): the maximum of the DIT (Depth of Inheritance Tree) values for each class in a class diagram. The DIT value for a class within a generalisation hierarchy is the longest path from the class to the root of the hierarchy.
8. Number of attributes (NA): the total number of attributes.
9. Number of methods (LOCMETHOD): the total number of methods.
10. Number of private attributes (NPRIVFIELD) : number of private attributes in a specific class

Our detection solution selects, from this exhaustive list, the best metrics combination that detects different defect types. In the next section, we emphasize the specific problems that are addressed by our detection approach.

2.3 Problem Statement

A tool supporting the detection and correction of design defects at the model level may be of great value for novice designers as well as experimented ones when refactoring existing models. However there are many open and challenging issues that we must address when building such a tool. Some of these open issues were introduced in (Kessentini et al., 2011b). We summarize these issues in the following.

In the current state of art, there is no consensus on what makes a particular design fragment a bad design. Even if we detect some design form that we defined as “suspicious”, we cannot say for sure that it is a defect (El-Boussaidi and Mili, 2011). Asserting that a suspicious design fragment is actually a design defect depends on the context. For example, a “Log” class responsible for maintaining a log of events, used by a large number of classes, is a common and acceptable practice. However, from a strict defect definition, it can be considered as a class with abnormally large coupling.

Furthermore even for the design defects that are commonly recognized in the literature such as the Blob, deciding which classes are Blob candidates depends on the designer’s interpretation. This also depends on the detection thresholds set by the designer when dealing with quantitative information. For example, the Blob detection involves information such as class size. Although we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given context could be considered average in another.

The last issue is related to the usefulness of detecting and returning long lists of defect candidates. In these cases, a designer needs to assess the defect candidates, select true positives that must be fixed and reject false positives. This can be a fastidious task and not always profitable.

In addition to these issues, manually defining the rules that detect all targeted design defects can be a time- consuming and an error-prone process.

2.4 Heuristic Search for Model Refactoring

2.4.1 Overview

To address the above mentioned issues, we propose an approach that exploits examples of model defects and a heuristic search technique to automatically build rules that detect defects

in models. The general structure of our approach is introduced in Figure 2.1.

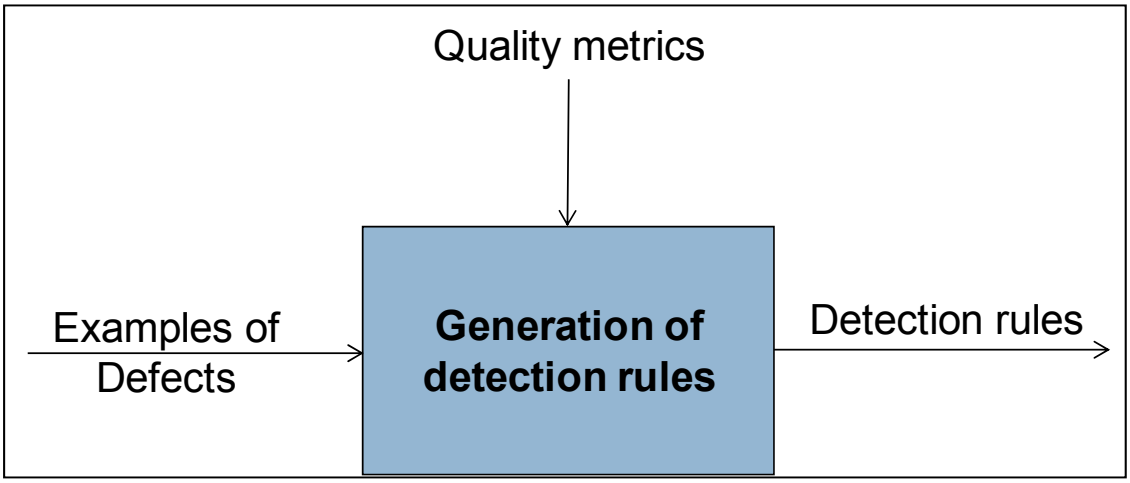


Figure 2.1 Overview of the approach

In our approach, knowledge from defect examples is used to generate detection rules. The detection algorithm takes as inputs a base (i.e. a set) of defect examples, and takes as controlling parameters a set of quality metrics (the expressions and the usefulness of these metrics were defined and discussed in the literature (Fenton and Pfleeger, 1998)). This algorithm generates as output a set of rules. The rule generation process combines quality metrics (and their threshold values) within rule expressions. Consequently, a solution to the defect detection problem is a set of rules that best detect the defects of the base of examples. For example, the following rule states that a class c having more than 10 attributes and 20 methods is considered as a blob defect:

$$R1: IF\ NA(c) \geq 10\ AND\ NMD(c) \geq 20\ THEN\ \ Blob \tag{2.1}$$

In this example of a rule, the number of attributes (NA) and the number of methods (NMD) of a class correspond to two quality metrics that are used to detect a blob defect. A class will be detected as a blob whenever both thresholds of 10 attributes and 20 methods are exceeded. Defect examples are in general available in repositories of new model projects under development, or previous projects under maintenance. Defects are generally documented as

part of the maintenance activity, and can be found in version control logs, incident reports, and inspection reports. The use of such examples has many benefits. First, it allows deriving defect detection rules that are closer to, and more respectful of, the designing “traditions” of model development teams in particular companies. These rules will be more precise and more context faithful, yet almost without loss of genericity, than more general rules, generated independently of any context. Second, it solves the problem of defining the values of the detection thresholds since these values will be found during the rule generation process. These thresholds will then correspond more closely to the company best practices. Finally, learning from examples allows reducing the list of detected defect candidates.

The rule generation process is executed periodically over large periods of time using the base of examples. The generated rules are used to detect the defects of any system that is required to be evaluated (in the sense of defect detection and correction). The rules generation step needs to be re-executed only if the base of examples is updated with new defect instances.
In the detection step, our approach assigns a threshold value randomly to each metric, and combines these threshold values using logical expressions (union OR; intersection AND) to create rules. The number m of possible threshold values is usually very large. The rules generation process consists of finding the best combination between n metrics. In this context, the number NR of possible combinations that have to be explored is given by:

$$NR = (n!)^m \tag{2.2}$$

This value quickly becomes huge. For example, a list of 5 metrics with 6 possible thresholds necessitates the evaluation of up to 1206 combinations.
Consequently, the rule generation process is a combinatorial optimization problem. Due to the huge number of possible combinations, a deterministic search is not practical, and the use of a heuristic search is warranted. To explore the search space, we use a global heuristic search by means of Genetic Programming (Koza, 1992). This algorithm will be detailed in next section.

2.4.2 Heuristic Search Using Genetic Programming

In this section we give an overview of Genetic programming (GP) and we describe how GP can be used to generate rules to detect design defects.

GP is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution (Koza, 1992). The basic idea behind GP is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem. In GP, a solution is a (computer) program which is usually represented as a tree, where the internal nodes are functions and the leaf nodes are terminal symbols. Both the function set and the terminal set must contain symbols that are appropriate for the target problem. For instance, the function set can contain arithmetic operators, logic operators, mathematical functions, etc; whereas the terminal set can contain the variables (attributes) of the target problem.

Each individual (i.e. a solution) of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem.

Exploration of the search space is achieved by selecting individuals (in the current population) that have the highest fitness values and evolving them using genetic operators, such as crossover and mutation. The crossover operator insures generation of new children, or offspring, based on parent individuals. The crossover operator allows transmission of the features of the best fitted parent individuals to new individuals. This is usually achieved by replacing a randomly selected sub tree of one parent individual with a randomly chosen sub tree from another parent individual to obtain one child. A second child is obtained by inverting parents. Finally, mutation operator is applied, with a probability which is usually inversely proportional to its fitness value, to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping local optima found during the search.

Once selection, mutation and crossover have been applied according to given probabilities, individuals of the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. This criterion usually corresponds to a fixed number of generations. The result of GP (the best solution found) is the fittest individual produced along all generations.

Hence to apply GP to a specific problem, the following elements have to be defined:

1. Representation of the individuals,
2. Creation of a population of individuals,
3. Evaluation of individuals using a fitness function,
4. Selection of the (best) individuals to transmit from one generation to another,
5. Creation of new individuals using genetic operators (crossover and mutation) to explore the search space,
6. Generation of a new population.

A high level view of our adaptation of GP to the defect detection problem is introduced by Algorithm 2.1. As this algorithm shows, it takes as input a set of quality metrics and a set of defect examples that were manually detected in some systems, and finds a solution, which corresponds to the set of detection rules that best detect the defects in the base of examples.

```
Input: Set of quality metrics
Input: Set of defect examples
Output: Detection rules

1     I:= rules(R, Defect_Type)
2     P:= set_of(I)
3     initial_population(P, Max_size)
4     repeat
5           for all I P do
6                 detected_defects := execute_rules(R)
7                 fitness(I) := compare(detected_defects,   defect_examples)
8           end for
9           best_solution := best_fitness(I);
10          P := generate_new_population(P)
11          it:=it+1;
12    until it=max_it
13    return best_solution
```

Algorithm 2.1 High-level pseudo-code for GP adaptation to our problem

Lines 1–3 construct an initial GP population, which is a set of individuals that stand for possible solutions representing detection rules. Lines 4–13 encode the main GP loop, which explores the search space and constructs new individuals by combining metrics within rules. During each iteration, we evaluate the quality of each individual in the population, and save the individual having the best fitness (line 9). We generate a new population (p+1) of individuals (line 10) by iteratively selecting pairs of parent individuals from population p and applying the crossover operator to them; each pair of parent individuals produces two children (new solutions). We include both the parent and child variants in the new population. Then we apply the mutation operator, with a probability score, for both parent and child to ensure the solution diversity; this produces the population for the next generation. The algorithm terminates when the termination criterion (maximum iteration

number) is met, and returns the best set of detection rules (best solution found during all iterations).

2.4.3 Heuristic Search Adaptation

The following three subsections describe more precisely our adaption of GP to the defect detection problem. To illustrate this adaption, we use a class diagram as a model to evaluate. Thus, the base of examples is a set of defect examples in a class diagram.

2.4.3.1 Individual Representation

An individual is a set of IF – THEN rules. For example, Figure 2.2 shows an individual (i.e. a solution) composed of three rules.

R1: IF (LOCCLASS(c) ≥ 1500 AND LOCMETHOD(m,c) ≥ 129) OR (NMD(c) ≥ 100) THEN BLOB(c)
R2: IF (LOCMETHOD(m,c) ≥ 151) THEN Poor Usage of Abstract Class (c)
R3: IF (NPRIVFIELD(c) ≥ 7 AND NMD(c) = 16) THEN Functional Decomposition (c)

Figure 2.2 Rule interpretation of an individual

A detection rule has the following structure:

IF “Combination of metrics with their threshold values” **THEN** “Defect type”

The IF clause describes the conditions or situations under which a defect type is detected. These conditions correspond to logical expressions that combine some metrics and their threshold values using logic operators (AND, OR). If some of these conditions are satisfied by a class, then this class is detected as the defect figuring in the THEN clause of the rule. Consequently, THEN clauses highlight the defect types to be detected. We will have as many

rules as types of defects to be detected. In our case, mainly for illustrative reasons, and without loss of generality, we focus on the detection of three defect types, namely blob, poor usage of abstract class and functional decomposition. Consequently, as it is shown in Figure 2.2, we have three rules, R1 to detect blobs, R2 to detect poor usage of abstract class, and R3 to detect functional decomposition.

One of the most suitable computer representations of rules is based on the use of trees (Davis et al., 1977). In our case, an individual is represented as a tree which is composed of two types of nodes: terminals and functions. The terminals (leaf nodes of a tree) correspond to different quality metrics with their threshold values. The functions that glue these metrics correspond to logical operators, which are Union (OR) and Intersection (AND).

Consequently, the three representation of the individual of Figure 2.2 is shown in Figure 2.3. This tree representation corresponds to an OR composition of three sub-trees, each sub tree representing a rule: R1 OR R2 OR R3.

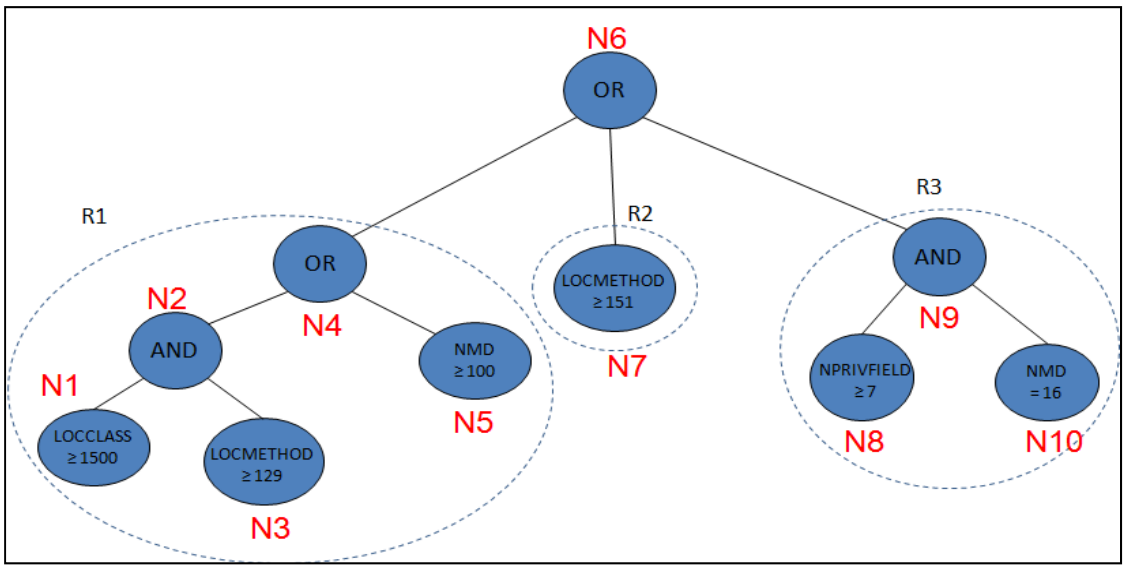


Figure 2.3 A tree representation of an individual

For instance, the first rule R1 is represented as a sub-tree of nodes starting at the branch (N1 – N5) of the individual tree representation of Figure 2.3. Since this rule is dedicated to detect

blob defects, we know that the branch (N1 – N5) of the tree will figure out the THEN clause of the rule. Consequently, there is no need to add the defect type as a node in the sub-tree dedicated to a rule.

2.4.3.2 Generation of an Initial population

To generate an initial population, we start by defining the maximum tree length including the number of nodes and levels. These parameters can be specified either by the user or randomly. Thus, the individuals have different tree length (structure). Then, for each individual we randomly assign:

1. One metric, with its threshold value, to each leaf node
2. A logic operator (AND, OR) to each function node

The root (head) of the tree is unchanged. Since any metric combination is possible and correct semantically, we do not need to define some conditions to verify when generating an individual. However, we need to ensure that the threshold values for each metric are correct (domain).

2.4.3.3 Genetic Operators

Selection

To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) (Koza, 1992), in which the probability of selection of an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select 50% of individuals from population p for the new population p+1. These (population_size/2) selected individuals will “give birth” to another (population_size/2) new individuals using crossover operator.

Crossover

Two parent individuals are selected and a node is picked on each one. Then crossover swaps the nodes and their relative sub trees from one parent to the other. The crossover operator can be applied only on parents having the same type of defect to detect. Each child thus combines information from both parents.

Figure 2.4 shows an example of the crossover process. In fact, the rule R1 and a rule RI1 from another individual (solution) are combined to generate two new rules. The right sub tree of R1 is swapped with the left sub tree of RI1.

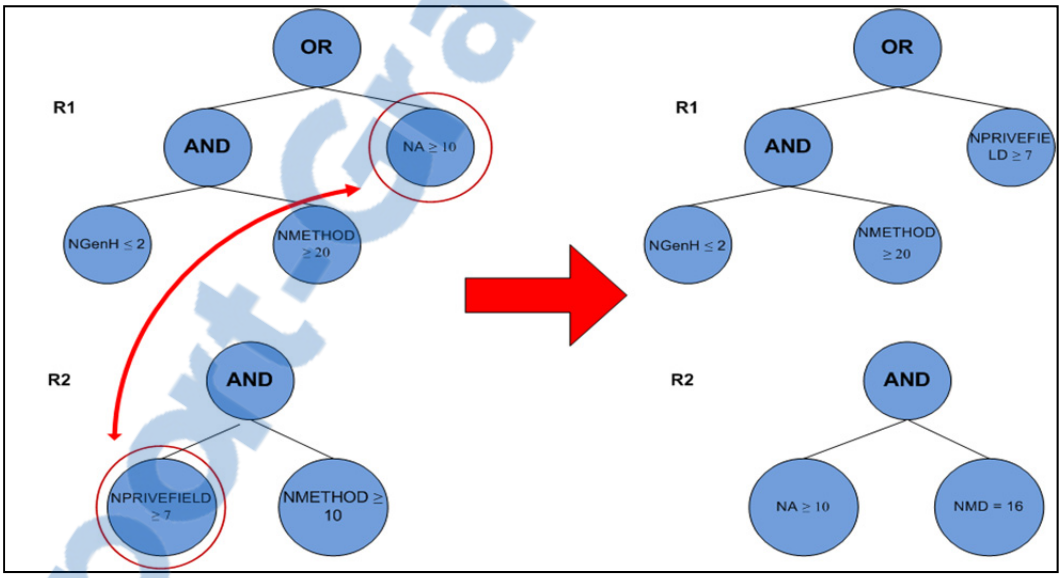


Figure 2.4 Crossover operator

As result, after applying the cross operator the new rule R1 to detect blob will be:
R1: IF (LOCCLASS(c) ≥ 1500 AND LOCMETHOD(m,c) ≥ 129)) OR (NPRIVFIELD(c) ≥ 7) Then blob(c)

Mutation

The mutation operator can be applied either to function or terminal nodes. This operator can modify one or many nodes. Given a selected individual, the mutation operator first randomly

selects a node in the tree representation of the individual. Then, if the selected node is a terminal (threshold value of a quality metric), it is replaced by another terminal. The new terminal either corresponds to a threshold value of the same metric or the metric is changed and a threshold value is randomly fixed. If the selected node is a function (AND operator for example), it is replaced by a new function (i.e. AND becomes OR). If a tree mutation is to be carried out, the node and its sub trees are replaced by a new randomly generated sub tree.

To illustrate the mutation process, consider again the example that corresponds to a candidate rule to detect blob defects. Figure 2.5 illustrates the effect of a mutation that deletes node NMD, leading to the automatic deletion of node OR (no left sub tree), and that replaces node LOCMETHOD by node NPRIVFIELD with a new threshold value. Thus, after applying the mutation operator the new rule R1 to detect blob will be:

R1: IF (LOCCLASS(c) ≥ 1500 AND NPRIVFIELD(c) ≥ 14)) THEN blob(c)

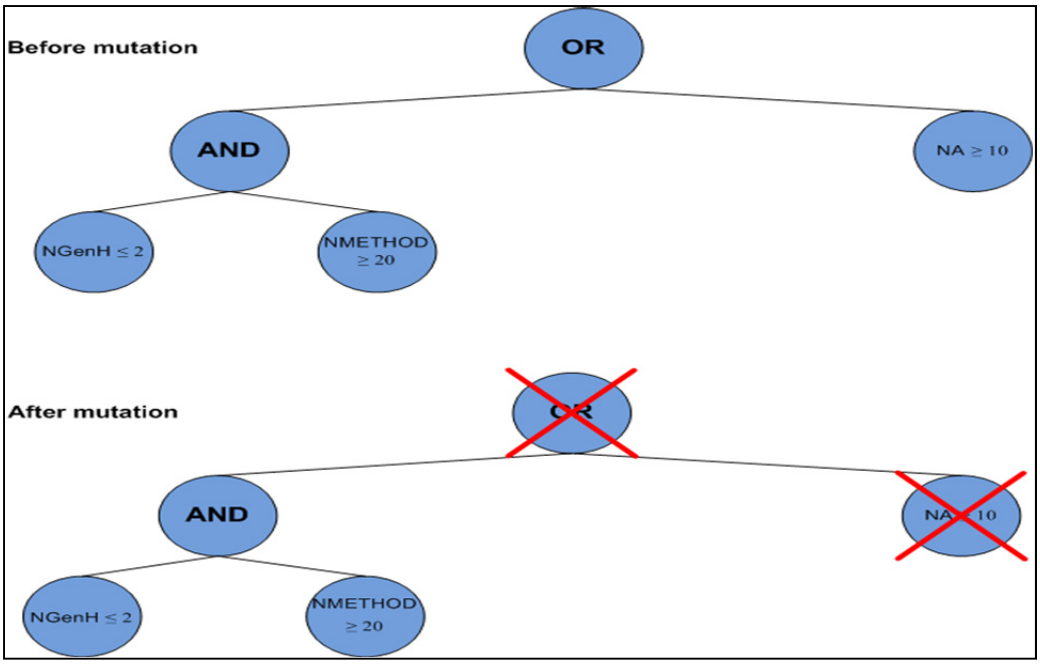


Figure 2.5 Mutation operator

2.4.3.4 Decoding of an Individual

The quality of an individual is proportional to the quality of the different detection rules composing it. In fact, the execution of these rules, on the different projects extracted from the base of examples (see Figure 2.6), detect various classes as defects. Then, the quality of a solution (set of rules) is determined with respect to the number of detected defects in comparison to the expected ones in the base of examples. In other words, the best set of rules is the one that detects the maximum number of defects.

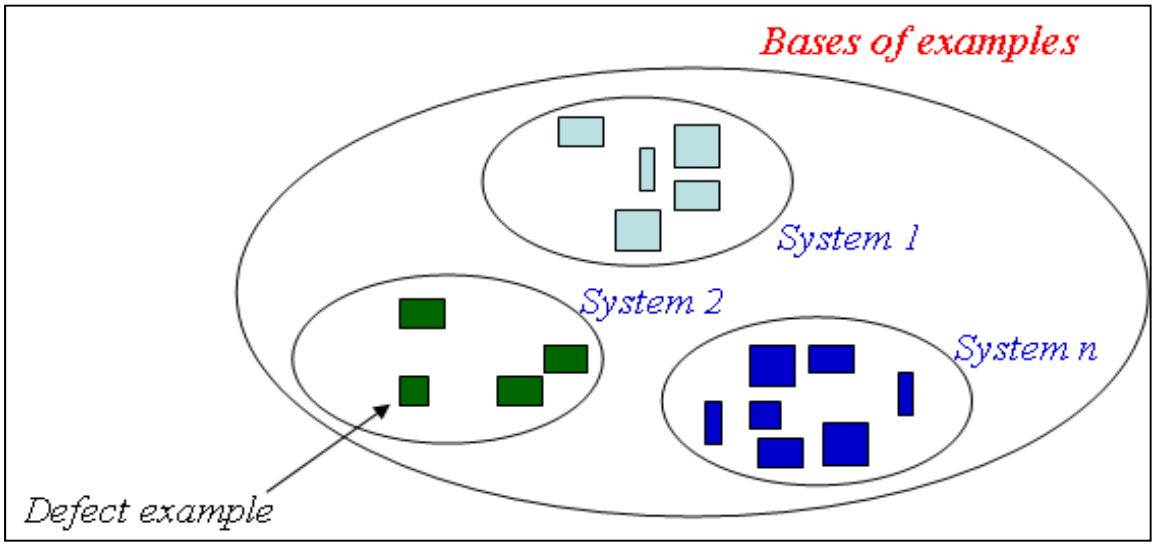


Figure 2.6 Base of examples

Consider, for example, a base of defect examples having three classes X, W, T that are considered respectively as blob, functional decomposition and another blob. Consider an individual (a solution) that contains different rules that detect only X as blob. In this case, the quality of this solution will have a value of 1/3 = 0.33 (only one detected defect over three expected ones).

The encoding of an individual should be formalized as a mathematical function called «fitness function». The fitness function quantifies the quality of the generated rules. The goal

is to define an efficient and simple (in the sense not computationally expensive) fitness function in order to reduce the computational complexity.

As discussed in section 2.2, the fitness function aims to maximize the number of detected defects in comparison to the expected ones in the base of examples. In this context, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f_{norm} = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2} \tag{2.3}$$

where t is the number of defects in the base of examples, p is the number of detected classes with defects, and a_i has value 1 if the i^{th} detected class exists in the base of examples (with the same defect type), and value 0 otherwise.

To illustrate the fitness function, we consider a base of examples containing one system evaluated manually. In this system, six (6) classes are subject to three (3) types of defects as shown in Table 2.1.

Table 2.1 Defects Example

Class	**Blob**	**Functional decomposition**	**Poor usage of abstract class**
Student		X	
Person		X	
University		X	
Course	X		
Classroom			X
Administration	X		

Table 2.2 lists the classes that were detected after executing the solution generating the rules R1, R2 and R3 of Figure 2.2.

Table 2.2 Detected classes

Class	Blob	Functional decomposition	Poor usage of abstract class
Person		X	
Classroom	X		
Professor		X	

Thus, only one class corresponds to a true defect (Person). Classroom is a defect but the type is wrong and Professor is not a defect. The fitness function has the value:

$$f_{norm} = \frac{\frac{1}{3} + \frac{1}{6}}{2} = 0.25 \tag{2.4}$$

with t=6 (only one defect is detected over 6 expected defects), and p=3 (3 defects were detected but only one corresponds to a defect in the base of examples).

2.5 Validation

In this section, we describe our experimental setup and present the results of an exploratory study.

2.5.1 Experimental settings

The goal of the experiment is to evaluate the efficiency of our approach for the detection of design defects in UML class diagrams. In particular the experiment aimed at answering the following research questions:

RQ1: To what extent can the proposed approach detect design defects?
RQ2: What types of defects does it locate correctly?

To answer RQ1, we used an existing corpus of known design defects (Moha et al., 2010) to evaluate the precision and recall of our approach. To answer RQ2, we investigated the type

of defects that were found. We used two open-source Java projects to perform our experiments: GanttProject (Gantt for short) v1.10.2, and LOG4J v1.2.1. We chose the LOG4J and Gantt libraries because they are medium-sized open-source projects and were analyzed in related work. The version of Gantt studied was known to be of poor quality, which has led to a new major revised version. LOG4J, on the other hand, has been actively developed over the past 10 years. We used Visual Paradigm tool (Paradigm, 2008) to generate class diagrams from these two open-source projects. Table 2.3 provides some relevant information about these projects.

Table 2.3 Program statistics

Systems	**Number of classes**
GanttProject v1.10.2	245
LOG4J v1.2.1	227

We asked a group of graduate students to analyze the libraries to tag instances of specific defects (blob, functional decomposition and Poor usage of abstract class) to validate our detection technique. Furthermore, we combined our manual inspection with the one proposed by Tiberghien et al. (Tiberghien et al., 2007).

Figure 2.7 shows a screenshot of the tool we implemented to evaluate our approach. This tool takes as input a list of metrics, a base of defect examples and the project to be evaluated. It generates as output the optimal solution, i.e. the detection rules. The defects found by applying the optimal solution are then compared to those tagged by students. We used a 2-fold cross validation procedure. For each fold, one open source project is evaluated by using the other project as base of examples. For example, Gantt is analyzed using detection rules generated from some defect examples from LOG4J and vice-versa.

In the following subsection we report the number of defects detected, the number of true positives, the recall (number of true positives over the number of design defects) and the precision (ratio of true positives over the number detected) for every defect in LOG4J and Gantt.

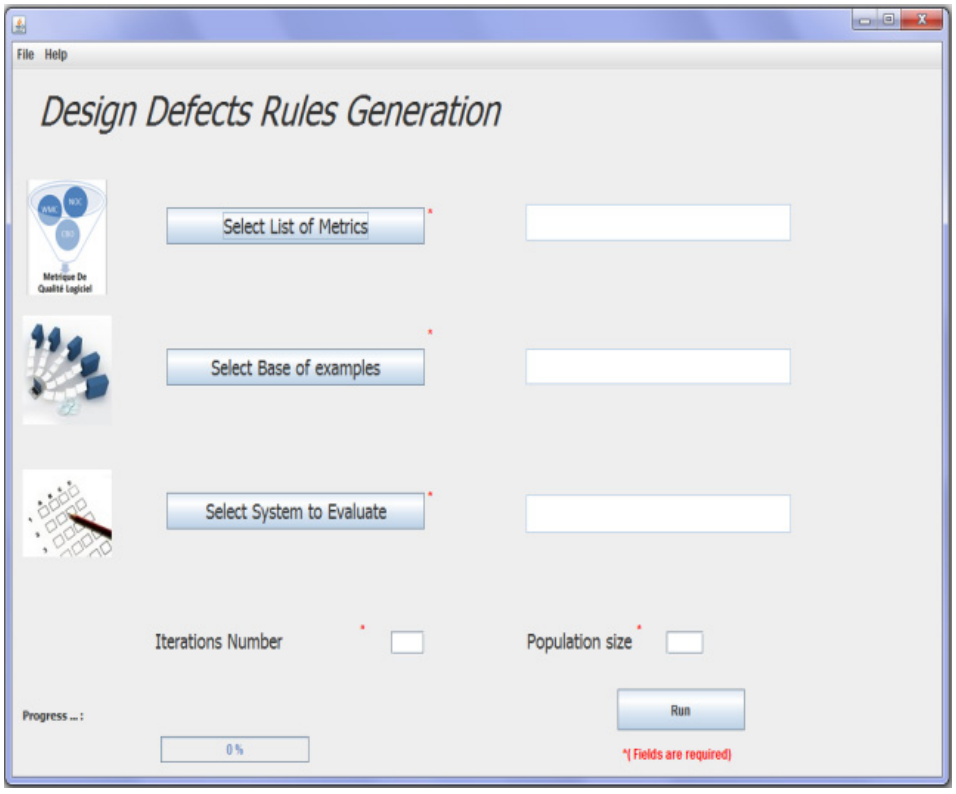
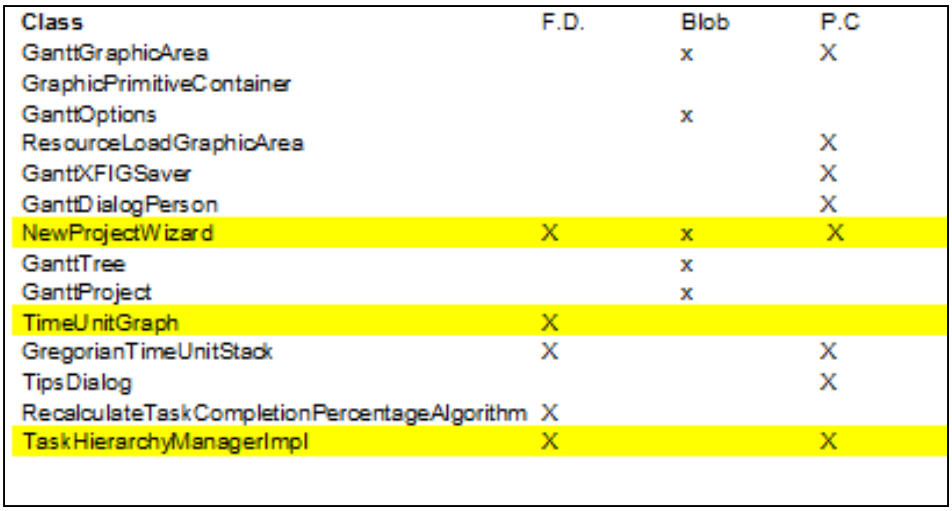


Figure 2.7 Rules generation tool

2.5.2 Results

Figure 2.8 and Table 2.4 summarize our findings. Figure 2.8 shows some detected defects, in Gantt class diagram, including only few false-positive ones (classes highlighted with a different color). For Gantt, the average defect detection precision was 94%. The average precision for LOG4J was 86%. In the context of this experiment, we can conclude that our technique was able to identify design defects with good precision and recall scores (answer to research question RQ1 above).



Class	F.D.	Blob	P.C
GanttGraphicArea		x	X
GraphicPrimitiveContainer			
GanttOptions		x	
ResourceLoadGraphicArea			X
GanttXFIGSaver			X
GanttDialogPerson			X
NewProjectWizard	X	x	X
GanttTree		x	
GanttProject		x	
TimeUnitGraph	X		
GregorianTimeUnitStack	X		X
TipsDialog			X
RecalculateTaskCompletionPercentageAlgorithm	X		
TaskHierarchyManagerImpl	X		X

Figure 2.8 Results obtained for the GanttProject

Table 2.4 Detection results

System	**Design defect**	**Precision**	**Recall**
GanttProject	Blob	100%	100%
	PC	83%	91%
	FD	91%	94%
LOG4J	Blob	87%	90%
	PC	84%	82%
	FD	66%	74%

We noticed that our technique does not have a bias towards the detection of specific anomaly types. In both projects, we had an almost equal distribution of each defect (answer to research question RQ2 above). On Gantt, the distribution was not as balanced, but this is principally due to the number of actual defects in the system.

One of the limitations of our proposal is the base of examples definition. In fact, the manual inspection of bad design practices can be a fastidious task. However, it can be argued that constituting such a set might require more work than identifying, specifying, and adapting rules. In our validation, we demonstrate that by using some open source projects directly, without any adaptation, our solution can be used out of the box and will produce good results for the detection of defects for the studied models.

Since we used a heuristic search technique, the detection results might vary depending on the rules generation process. In fact, the rules are randomly generated, though guided by a meta-heuristic. To ensure that our results are relatively stable, we compared the results of multiple executions for rules generation. We consequently believe that our technique is stable, since the precision and recall scores are approximately the same for different five executions.

In addition, it is important to contrast the results with the execution time because we used a heuristic search technique. We executed our algorithm on a standard desktop computer: Pentium CPU running at 2 GHz with 3GB of RAM. The execution time for rules generation with a number of iterations, as stopping criteria, fixed to 200 was less than three minutes (2min9s). This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of used metrics and the size of the base of examples.

2.6 Related Work

Several approaches tackled the problem of detecting and fixing design defects in software using different techniques. These techniques range from fully automatic detection and correction to guided manual inspection. However, the majority of these solutions are related to detect defects in the code level. The related work can be classified into three broad categories: rules-based detection-correction, detection and correction combination, and visual-based detection.

In the first category, Marinescu (Marinescu, 2004) defined a list of rules relying on metrics to detect what he calls design flaws of OO design at method, class and subsystem levels. Erni et al. (Erni and Lewerentz, 1996) use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, n-tuples of metrics expressing a quality criterion (e.g., modularity). The main limitation of the two previous contributions is the difficulty to manually define threshold values for metrics in the rules. To circumvent this problem, Moha et al. (Moha et al., 2010), in their DECOR approach, start by describing

defect symptoms using an abstract rule language. These descriptions involve different notions, such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions which results in an important rate of false positives. In our approach, the above-mentioned problems related to the use of rules and metrics do not arise. Indeed, the symptoms are not explicitly used, which reduces the manual adaptation/calibration effort.

The majority of existing approaches to automate refactoring activities are based on rules that can be expressed as assertions (invariants, pre- and post-condition), or graph transformation. The use of invariants has been proposed to detect parts of program that require refactoring by (Kataoka et al., 2001). Opdyke (Opdyke, 1992) suggest the use of pre- and postcondition with invariants to preserve the behavior of the software. All these conditions could be expressed in the form of rules. (Heckel, 1995) considers refactorings activities as graph production rules (programs expressed as graphs). However, a full specification of refactorings would require sometimes large number of rules. In addition, refactoring-rules sets have to be complete, consistent, non-redundant, and correct. Furthermore, we need to find the best sequence of applying these refactoring rules. In such situations, search-based techniques represent a good alternative. In (Kessentini et al., 2010), we have proposed another approach, based on search-based techniques, for the automatic detection of potential design defects in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. The two approaches are completely different. We use in (Kessentini et al., 2010) a good quality of examples in order to detect defects; however in this work we use defect examples to generate rules. In addition, this work is concerned with defects in the model level. Both works do not need a formal definition of defects to detect them.

In the second category of work, defects are not detected explicitly. They are so implicitly because the approaches refactor a system by detecting elements to change to improve the global quality. For example, in (O'Keeffe, 2008), defect detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the

improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximizes a function, which captures the variations of a set of metrics (Harman and Clark, 2004). The fact that the quality in terms of metrics is improved does not necessary mean that the changes make sense. The link between defect and correction is not obvious, which makes the inspection difficult for the maintainers.

The high rate of false positives generated by the automatic approaches encouraged other teams to explore semiautomatic solutions. These solutions took the form of visualization-based environments. The primary goal is to take advantage of the human ability to integrate complex contextual information in the detection process. Kothari et al. (Kothari et al., 2004) present a pattern-based framework for developing tool support to detect software anomalies by representing potential defects with different colors. Later, Dhambri et al. (Dhambri et al., 2008) propose a visualization-based approach to detect design anomalies by automatically detecting some symptoms and letting others to the human analyst. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Still, the visualization approach is not obvious when evaluating large-scale systems. Moreover, the information visualized is for the most part metric-based, meaning that complex relationships can still be difficult to detect. In our case, human intervention is needed only to provide defect examples.

2.7 Conclusion

In this article, we described a new solution to detect model-refactoring opportunities especially related to design defects. Existing work try to define different types of common design defects and describe symptoms to search for in order to locate the design defects. In our proposal, we have revealed that this knowledge is not necessary to perform the detection. Instead, we use examples of design defects and generic programming to generate defect detection rules. We obtained good performance by evaluating our solution to detect different defect types on large models extracted from open source projects.

As part of future work, we plan to extend our base of examples with additional badly-designed models in order to take into consideration more design contexts. In addition, we are working on the model refactoring step and to adapt the proposed approach to classify changes between different model versions as risky or not.

In the next chapter, we use examples of design defects to detect a set of design defects in a given model (class diagram) by adapting GA and we compare the results of this approach to the results obtained in the current chapter.

CHAPTER 3

A DESIGN DEFECT EXAMPLE IS WORTH A DOZEN DETECTION RULES

Adnane Ghannem, Ghizlane El Boussaidi[1] and Marouane Kessentini[2]
[1]Department of Software and IT Engineering, École de Technologie Supérieure,
1100 Notre-Dame West, Montreal, Quebec, (H3C 1K3) Canada
[2]Department of Computer and Information Science, University of Michigan - Dearborn
4901 Evergreen Road, Dearborn, MI 48128 USA

This paper has been accepted for publication
in Software Quality Journal

ABSTRACT

Design defects are symptoms of design decay which can lead to several maintenance problems. To detect these defects, most of existing research is based on the definition of rules that represent a combination of software metrics. These rules are sometimes not enough to detect design defects since it is difficult to find the best threshold values, the rules do not take into consideration the programming context and it is challenging to find the best combination of metrics. As an alternative, we propose in this paper to identify design defects using a genetic algorithm based on the similarity/distance between the system under study and a set of defect examples without the need to define detection rules. We tested our approach on four open-source systems to identify three potential design defects. The results of our experiments confirm the effectiveness of the proposed approach.

Keywords: Search-based software engineering, design defects, Detection by example, Genetic Algorithm.

3.1 Introduction

Model-driven engineering (MDE) is an approach to software development by which software is specified, designed, implemented and deployed through a series of models (Bull, 2008). MDE activities reduce the development and maintenance effort by analyzing and mainly

modifying systems at the model level instead of the code level. One of the main MDE activities is model maintenance defined as different modifications made on a model in order to improve his quality, adding new functionalities, detecting bad designed fragments, correcting them, and modifying the model, etc. (Marinescu, 2004). Due to the high cost related to these activities, automated solutions to improve model quality are a must.

To support maintenance and improve the quality of software, several approaches were proposed in the literature (e.g. (Du Bois et al., 2004; El-Boussaidi and Mili, 2011; Marinescu, 2004; Mens et al., 2007a; Moha et al., 2010; Ragnhild et al., 2007; Van Kempen et al., 2005; Zhang et al., 2005)). Most of these approaches focus on detecting and correcting design defects. To do so, they rely on declarative rules that are manually defined; these rules are specified using metrics that embody the symptoms related to the design defect. For example, the design defect called Blob (Brown et al., 1998) is characterized by symptoms like a high number of methods, attributes and relations with many Data-Classes. Nevertheless, there is no consensus on what makes a particular design fragment a design defect. Furthermore, for most common design defects, defining appropriate threshold values for the related metrics is not obvious. For example, a rule that detects Blob classes involves metrics related to the class size (e.g., number of methods). Although we can easily calculate these metrics, appropriate threshold values are not trivial to define. In addition, existing work has, for the most part, focused on detecting and correcting (refactorings) design defects at the source code level. Very few approaches tackled this problem at the model level (e.g., (El-Boussaidi and Mili, 2011; Mens et al., 2007a; Zhang et al., 2005)). Most of the model-based approaches are based on rules that can be expressed as assertions (i.e., invariants, pre-and post-condition) (Ragnhild et al., 2007; Van Kempen et al., 2005), or graph transformations targeting refactoring operations in general (e.g., (Du Bois et al., 2004; El-Boussaidi and Mili, 2011)) or refactorings related to design patterns' applications (e.g., (El-Boussaidi and Mili, 2011)). However, a complete specification of defects detection and correction requires an important number of rules and these rules must be complete, consistent, non-redundant and correct.

In this work, we start from two main observations: 1) design defects detection rules are difficult to define; and 2) they do not capitalize on defect repositories that may be available

in many companies where defects in projects under development are manually identified, corrected and documented. Based on these observations, we propose a by example approach that exploits existing examples of defects to overcome the problems related to explicitly defining detection rules. Our approach takes as inputs an initial model and a base of defect examples, and takes as controlling parameters a set of software metrics and it generates a design defects set detected in the initial model. To this end, we used a population-based meta-heuristic search based on Genetic Algorithms (GA) (Goldberg, 1989). In the context of this paper, we focus on detecting defects in UML class diagrams. Our approach is evaluated on four large open source systems, and aimed at investigating to what extent the use of the base of examples of design defects improve the automation of detection.

The primary contributions of the paper can be summarized as follows:

1. We introduce a detection approach based on the use of design defect examples. Our proposal does not require an explicit definition of detection rules; and thus it does not require a specification of the metrics to use or their related threshold values.

2. We report the results of an evaluation of our approach; we used design defect examples extracted from four object-oriented open source projects. We applied an four-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining three systems as bases of examples. The average values of precision and recall computed from 31 executions on each project are 95% and 76% respectively which allows us to say that the obtained results are promising. The effectiveness of our approach is also assessed using a comparative study between our approach and two other approaches.

The remainder of this paper develops our proposals and details how they are achieved. Therefore, the paper is structured as follows. Section 3.2 is dedicated to the background and problem statement related to our approach. Section 3.3 presents the overall approach and the details of our adaptation of the genetic algorithm to the problem of detecting design defects

in UML class diagrams. Section 3.4 reports on the experimental settings and results. Related works are discussed in section 3.5 and we conclude and outline some future directions to our work in section 3.6.

3.2 Background and Problem Statement

3.2.1 Design Defects

We focus in this paper on the detection of a specific type of design defect to improve model quality. Design defects, also called design anomalies, refer to design situations that adversely affect the development of models (Brown et al., 1998). Different types of defects, presenting a variety of symptoms, have been studied in the intent of facilitating their detection and suggesting improvement solutions. In (Fowler and Beck, 1999), they define a set of symptoms of common defects. These include large classes, feature envy, long parameter lists, and lazy classes. Each defect type is accompanied by refactoring suggestions to correct the defect. Brown et al. (Brown et al., 1998) define another category of design defects that are documented in the literature, and named anti-patterns.

In our approach, we focus on the detection of some defects that can appear at the model level and especially in class diagrams. We choose from (Brown et al., 1998) three important defects that can be detected in class diagrams:

1. Blob which is found in designs where one large class monopolizes the behavior of a system (or part of it), and other classes primarily encapsulate data.
2. Functional decomposition (FD): It occurs when a class is designed with the intent of performing a single function. This is found in class diagrams produced by non-experienced object-oriented developers.
3. Data Class (DC): It encapsulates only data. The only methods that are defined by this class are the getters and the setters.

3.2.2 Software Metrics

Software metrics provide useful information that help assessing the level of conformance of a software system to a desired quality such as evolvability and reusability (Fenton and Pfleeger, 1998). Metrics can also help detecting some similarities between software systems. The most widely used metrics for class diagrams are the ones defined by Genero et al. (Genero et al., 2002). In the context of our approach, we used the eleven (11) metrics defined in (Genero et al., 2002) to which we have added a set of simple metrics (e.g., number of private methods in a class, number of public methods in a class) that we have defined for our needs. The metrics configuration for the experiments reported here consisted of the sixteen software metrics described below in Table 3.1. All these metrics are related to the class entity which is the main entity in a class diagram. Some of these metrics represent statistical information (e.g. number of methods, attributes, etc.) and others give information about the position of the class through its relationships with the other classes of the model (e.g. number of associations). All these metrics have a strong link with the design defects presented in the previous section.

Table 3.1 Considered metrics in our approach

Ref	***Metric Description***
NA	The total number of attributes per class.
NPvA	The total number of private attributes per class.
NPbA	The total number of public attributes per class.
NProtA	The total number of protected attributes per class.
NM	The total number of methods per class.
NPvM	The total number of private methods per class.
NPbM	The total number of public methods per class.
NPrtM	The total number of protected methods per class.
NAss	The total number of associations.
NAgg	The total number of aggregation relationships.
NDep	The total number of dependency relationships.
NGen	The total number of generalisation relationships (each parent-child pair in a generalization relationship).
NAggH	The total number of aggregation hierarchies.

NGenH	The total number of generalisation hierarchies.
DIT	The DIT value for a class within a generalisation hierarchy is the longest path from the class to the root of the hierarchy.
HAgg	The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.

3.2.3 Problem Statement

A tool supporting the detection and correction of design defects at the model level may be of great value for novice designers as well as experimented ones when refactoring existing models. However, there are many open challenging issues that we must address when building such a tool. Some of these open issues were introduced in (Kessentini et al., 2011b).

In the current state of art, there is no consensus on what makes a particular design fragment a bad design. Even if we detect some design form that we defined as "suspicious", we cannot say for sure that it is a defect (El-Boussaidi and Mili, 2011). Asserting that a suspicious design fragment is actually a design defect depends on the context. For example, a «Log» class responsible for maintaining a log of events, used by a large number of classes, is a common and acceptable practice. However, from a strict defect definition, it can be considered as a class with abnormally large coupling. Furthermore, even for the design defects that are commonly recognized in the literature such as the Blob, deciding which classes are Blob candidates depends on the designer's interpretation. This also depends on the detection thresholds set by the designer when dealing with quantitative information. For example, the Blob detection involves information such as class size. Although we can measure the size of a class, an appropriate threshold value is not trivial to define. A class considered large in a given context could be considered average in another. Another issue is related to the usefulness of detecting and returning long lists of defect candidates. In these cases, a designer needs to assess the defect candidates, select true positives that must be fixed and reject false positives. This can be a fastidious task and not always profitable. In addition to these issues, manually defining the rules that detect all targeted design defects can be a time-consuming and an error-prone process. Finally, it is difficult to generalize the detection

rules from a set of defect examples. Therefore, we argue that it is more efficient to rely on similarities between the software under analysis and existing defect examples to detect design defects in this software. This idea is the foundation of the approach proposed in this paper.

3.3 A Search Based Approach to Detecting Design Defects

The approach proposed in this paper exploits examples of design defects and a heuristic search technique to automatically detect design defects on a given model and specifically in class diagrams. Our detection approach takes as inputs an initial model and a base (i.e. a set) of defect examples, and takes as controlling parameters a set of software metrics. These metrics were presented above in Table 1 and their expressiveness and usefulness were discussed in the literature (Genero et al., 2002). The approach generates a set of design defects detected in the initial model. In the following subsection, we describe in details how we encoded the design defects detection problem using the Genetic Algorithm (GA) (Koza, 1992).

3.3.1 Adaptation of the Genetic Algorithm to Design Defects Detection

GA is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution (Koza, 1992). A high-level view of our adaptation of GA to the design defect detection problem is given by Algorithm 3.1 which takes as input an initial model, a set of software metrics and a set of design defects examples. The output is the set of design defects that were detected in the initial model.

```
Input: Initial Model (IM)
Input: Set of quality metrics
Input: Set of Design Defects examples
Output: A set of Design Defects (DD) detected in IM
1: I:= set_of (CIM, CBE, DD detected on  CBE)
2: P:= set_of (I)
3: initial_population (P, Max_size)
4: Repeat
5:    for all I P do
7:       Fitness(I) : = {f1(I) + f2(I)}/2
8:    end for
9:    best_solution := best_fitness(I);
10:  P := generate_new_population(P)
11:  it:=it+1;
12: Until it = max_it or fitness(best_solution) =1;
13: return best_solution
```

Algorithm 3.1 High-level pseudo-code for GA adaptation to our problem

Lines 1–3 construct an initial population, which is a set of individuals that stand for possible solutions representing a set of design defects that may be detected in the classes of the initial model. An individual is a set of triplets; a triplet is called a block and it contains a class of the initial model denoted as CIM, a class of the base of examples denoted as CBE, and a design defect (DD) detected in the CBE. To generate an initial population, we start by defining the maximum individual size in terms of a maximum number of blocks composing an individual. This parameter can be specified either by the user or randomly. Thus, the individuals have different sizes. Then, for each individual, the blocks are randomly built; i.e., a block is composed by the triplet (CIM, CBE, DD) where a class (CIM) from the initial model is

randomly matched to a class (CBE) from the base of examples and a design defect (DD) present in the CBE.

Individuals' representation is explained in more detail in section 3.3.2. Lines 4–12 encode the main GA loop, which explores the search space and constructs new individuals by changing the matched pairs (CIM, CBE) in blocks. During each iteration, we evaluate the quality of each individual in the population. To do so, we use a fitness function defined as an average of two functions f1 and f2. f1 computes the similarities between the classes CMI and CBE of each block composing the individual while f2 computes the ratio of the individual size by the maximum individual size (line 7). Computation of these two functions and the fitness function of an individual is described in more detail in section 3.3.4. Then we save the individual having the best fitness (line 9). In line 10, we generate a new population (p+1) of individuals from the current population by selecting 50% of the best fitted individuals from population p and generating the other 50% of the new population by applying the crossover operator to the selected individuals; i.e., each pair of selected individuals, called parents, produces two children (new solutions). Then we apply the mutation operator, with a probability, for both parents and children to ensure the solution diversity; this produces the population for the next generation. The mutation probability specifies how often parts of an individual will mutate. Selection, crossover and mutation are described in details in section 3.3.3.

The algorithm stops when the termination criterion is met (Line 12) and returns the best solution found during all iterations (Line 13). The termination criteria can be a maximum number of iterations or the best fitness function value. However, the best fitness function value is difficult to predict and sometimes it takes very long time to converge towards this value. Hence, our algorithm is set to stop when it reaches the maximum iteration number or the best fitness function value. In the following subsections, we describe in details our adaption of GA to the design defect detection problem.

3.3.2 Individual representation

An individual is a set of blocks. A block contains three parts as shown by Figure 3.1: the first part contains the class CIM chosen from the initial model (model under analysis), the second part contains the class CBE from the base of examples that was matched to CIM, and finally the third part contains the design defect detected on CBE. An example of a solution (i.e., an individual) is given in Figure 3.2.

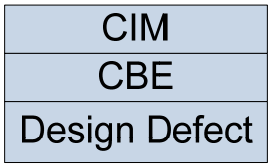


Figure 3.1 Block representation

Order	LineOrder	Product
Person	Teacher	Agency
Blob	Blob	FD

Figure 3.2 Individual representation

3.3.3 Genetic Operators

3.3.3.1 Selection

We used the stochastic universal sampling (SUS) (Koza, 1992) to select individuals that will undergo the crossover and mutation operators to produce a new population from the current one. In the SUS, the probability of selecting an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select 50% of individuals from population p for the new population p+1. These (population_size/2) selected individuals will be transmitted from the current generation to the new generation and they will «give-birth» to another (population_size/2) new individuals using crossover operator.

3.3.3.2 Crossover

For each crossover, two individuals are selected by applying the SUS selection (Koza, 1992). Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring P'1 and P'2 from the two selected parents P1 and P2. It is defined as follows: A random position, k, is selected. The first k blocks of P1 become the first k blocks of P'2. Similarly, the first k blocks of P2 become the first k blocks of P'1. The rest of blocks (from position k+1 until the end of the set) in each parent P1 and P2 are kept. For instance, Figure 3.3 illustrates the crossover operator applied to two individuals (parents) P1 and P2. The position k takes the value 2. The first two blocks of P1 become the first two blocks of P'2. Similarly, the first two blocks of P2 become the first two blocks of P'1.

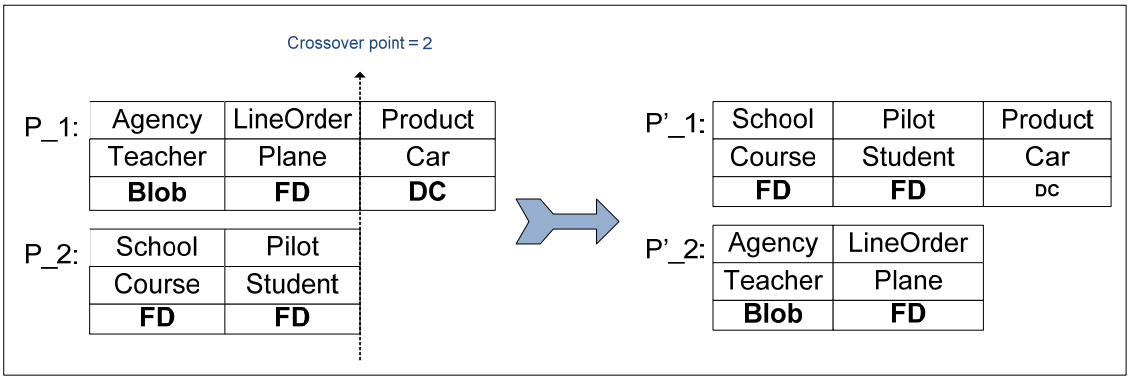


Figure 3.3 Crossover operator

3.3.3.3 Mutation

The mutation operator consists of randomly changing one or more dimensions (i.e., blocks) in the solution. Hence, given a selected individual, the mutation operator first randomly selects some blocks in the individual. Then the CBE of the selected block is replaced by another CBE chosen randomly from the base of examples. Figure 3.4 illustrates the effect of a mutation that replaced the design defect Blob detected in the class Teacher (initial model)

which is extracted from the class Agency (base of examples) by the design defect Data_Class (DC) extracted from the new matched class Taxes (base of examples).

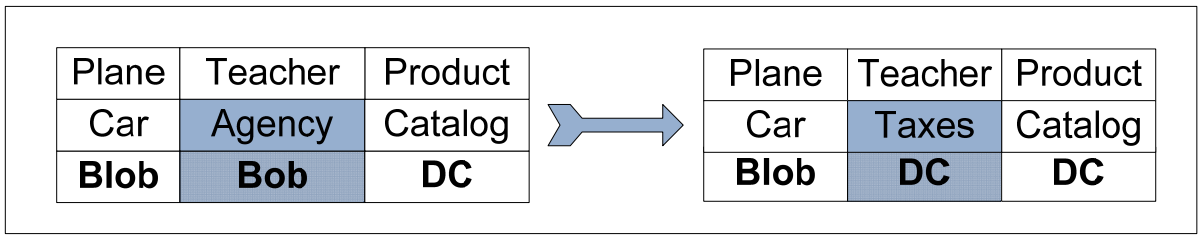


Figure 3.4 Mutation operator

3.3.4 Fitness function

The fitness function quantifies the quality of the generated individuals. The challenge is to define an efficient and simple fitness function in order to reduce the computational complexity. In our context, we want to exploit the similarities between the model under analysis and other existing models to infer the design defect that we must correct. Our intuition is that a candidate solution that displays a high similarity between the classes of the actual model and those chosen from the examples base should give the most accurate set of design defects. Hence, the fitness function aims to maximize the similarity between the classes of the model in comparison to the ones in the base of examples. In this context, we introduce first a similarity measure between two classes denoted by Similarity and defined by formulae 4.1 and 4.2.

$$Similarity\ (CIM, CBE) = \frac{1}{m}\sum_{i=1}^{m} Sim(CIM_i, CBE_i) \tag{3.1}$$

$$Sim\ (CIM_i, CBE_i) = \begin{cases} 1 & if\ CIM_i = CBE_i \\ 0 & if\ (CIM_i = 0\ and\ CBE_i \neq 0\ )\ or\ (CIM_i \neq 0\ and\ CBE_i = 0) \\ \frac{CIM_i}{CBE_i} & if\ CIM_i < CBE_i \\ \frac{CBE_i}{CIM_i} & if\ CBE_i < CIM_i \end{cases} \tag{3.2}$$

Where m is the number of metrics considered in this project. CIMi is the ith metric value of the class CIM in the initial model while CBEi is the ith metric value of the class CBE in the base of examples. Using the similarity between classes, we define the first component (f1) of the fitness function of a solution defined by the formula 4.3. We also add a second component (f2) of the fitness function to ensure the completeness of the solution defined by the formula 4.4; i.e., f2 takes into consideration the size of an individual in terms of the number of blocks compared to the maximum individual size. As discussed in section 3.3.1, the maximum individual size represents the maximum number of blocks composing an individual and it can be specified either by the user or randomly.

$$f_1 = \frac{1}{n}\sum_{j=1}^{n} Similarity(CIM_{Bj}, CBE_{Bj}) \tag{3.3}$$

$$f_2 = \frac{Individual\ size}{Maximum\ individual\ size} \tag{3.4}$$

Where n is the number of blocks in the solution and CIMBj and CBEBj are the classes composing the first two parts of the jth block of the solution. Finally, we define the fitness function of a solution, normalized in the range [0, 1], as denoted by the formula 4.5.

$$ff = \frac{f_1 + f_2}{2} \tag{3.5}$$

To illustrate how the fitness function is computed, consider as an example an individual I composed by two blocks. The first block matches the class Plane from the initial model to the class Catalog from the base of examples, while the second block matches the class Car from the initial model to the class Agency from the base of examples. In this example, the maximum individual size is set to 10 and we use five metrics. The values of these metrics are given for the classes composing the individual I in Table 3.2 (for classes from the initial model) and Table 3.3 (for classes from the base of examples).

Table 3.2 Classes from the initial model and their metrics values

CMI	NPvA	NPbA	NPvM	NAss	NGen
Plane	4	1	1	1	1
Car	2	2	0	1	0

Table 3.3 Classes from the base of examles and their metrics values

CBE	NPvA	NPbA	NPvM	NAss	NGen
Agency	2	1	0	3	0
Catalog	5	1	0	1	0

The fitness function of I is calculated as follows:

$$f_{1_I} = \frac{1}{2}\left[\frac{1}{5}\left[\left(\frac{4}{5} + 1 + 0 + 1 + 0\right) + \left(1 + \frac{1}{2} + 1 + \frac{1}{3} + 1\right)\right]\right] = 0{,}66$$

$$f_{2_I} = \frac{2}{10} = 0.2$$

$$ff = \frac{f_{1_I} + f_{2_I}}{2} = \frac{0.66+0.2}{2} = 0.43$$

3.4 Validation of the Approach

We implemented and tested the approach on four open source projects. In this section, we describe our experimental setup and we present the results of our experiment. We specifically discuss the results of our GA algorithm in terms of precision and recall.

3.4.1 Research Questions

The goal of the experiment is to evaluate the efficiency of our approach for the detection of design defects in UML class diagrams. Specifically, the experiment aimed at answering the following research questions:

RQ1: To what extent can the proposed approach detect design defects?
RQ2: What types of defects does it locate correctly?

To answer RQ1, we used an existing corpus of known design defects to evaluate the precision and recall of our approach. In the context of our study, the precision denotes the fraction of true design defects among the set of all detected defects. The recall indicates the fraction of correctly detected design defects among the set of expected defects (i.e., how many defects have not been missed). In general, the precision denotes the correctness of the approach (i.e., the probability that a detected defect is a true defect) and the recall denotes the completeness of the approach (i.e., the probability that an actually defect is detected). To answer RQ2, we investigated the type of defects that were detected by our approach.

3.4.2 Experimental Setup

We implemented our approach as an Eclipse plugin that takes as input the model under analysis (class diagram), a list of metrics and a base of examples. It generates as output the optimal solution; i.e., a set of design defects detected in the analyzed model. This plugin includes a parser which analyses Java source code and generates a predicate model that is manipulated by our GA algorithm. The predicate model was introduced in our previous work (Ghannem et al., 2013).

We used four open-source Java projects to perform our experiments. We chose theses open source projects because they are medium-sized open-source projects and were analysed in related work. Table 3.4 provides some relevant information about these projects including the number of classes and the number of design defects (i.e., Blob, Functional Decomposition and Data Class) existing in these projects. We used our parser to generate predicate models from the four selected projects. To build the base of examples, we completed the generated models by manually entering the design defects that were detected by related work (Kessentini et al., 2010; Moha et al., 2010) in these projects. We used a four-fold cross validation procedure. For each fold, one open source project is evaluated by using the

remaining three projects as base of examples. For example, Gantt is analyzed using LOG4J, ArgoUML and Xerces as base of examples and vice-versa. We also performed multiple executions of the approach on each of the 4 projects to ensure that the results of our approach are stable.

Table 3.4 Case Study setting

Project	# of classes	# of Blobs	# of FDs	# of DCs
GanttProject v1.10.2	245	10	17	10
LOG4J v1.2.1	227	3	11	5
ArgoUML v0.18.1	1267	29	37	41
Xerces v2.7	676	44	29	58

We report the number of defects detected, the number of true positives, the recall (number of true positives over the number of design defects) and the precision (ratio of true positives over the number detected), we determined the values of these indicators when using our algorithm for every defect in the four open source projects (for more detail see ANNEX I, p. 209).

To set the parameters of GA for the search strategies, we performed several tests and the final parameters' values were set to a minimum of 1000 iterations for the stopping criterion, to 2 as the minimum length of a solution in terms of number of block, and to size of the initial model as the maximum length of a solution. We also set the crossover probability to 0.9 and the mutation probability to 0.5. These values were obtained by trial and error. We selected a high mutation rate because it allows the continuous diversification of the population which discourage premature convergence to occur.

Finally, since we viewed the design defects detection problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (Intel i7 CPU running at 2.67 GHz with 8GB of RAM). The execution time for detection defects with a number of iterations 1000 was less than one minute. This indicates that our approach is reasonably

scalable from the performance standpoint. However, the execution time depends on the number of used metrics and the size of the base of examples.

3.4.3 Results and discussion

Figure 3.5 shows the precision results of multiple executions (31) of the approach for all the projects. The averages of precision are 93%, 100%, 94% and 94% for Gantt, Log4J, ARGOUML and Xerces, respectively. Similarly, Figure 3.6 shows the recall results of the 31 executions for all the projects. The averages of the recall are 76%, 76%, 75%, 76% for Gantt, Log4J, ARGOUML and Xerces, respectively. Generally, the high precision and recall average allows us to positively answer our first research question RQ1. Indeed, the precision average which is very high for all the projects (close to 100%) proves that all the design defects detected by our approach were true existing defects in the analyzed projects.

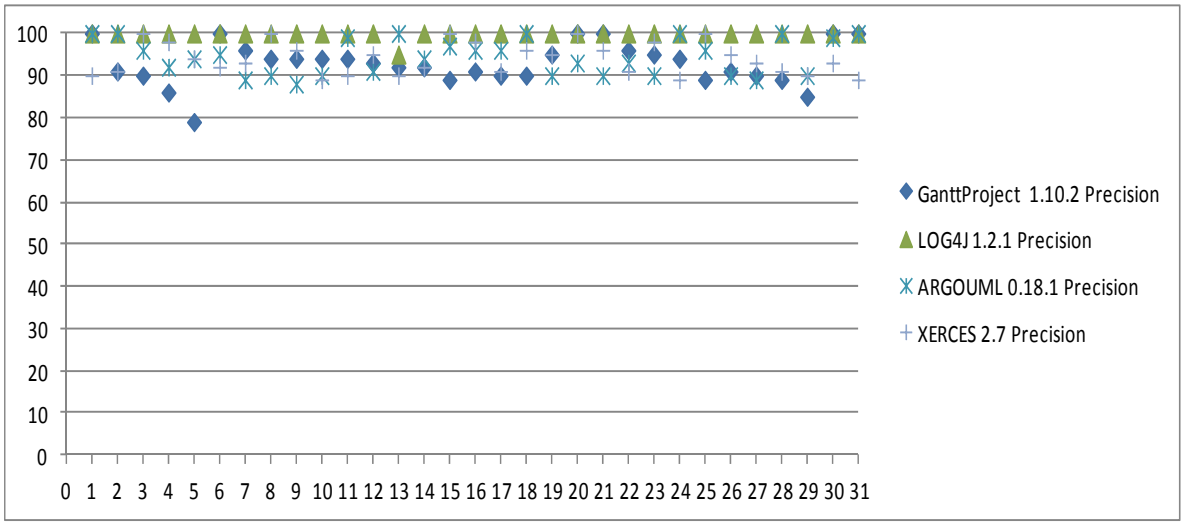


Figure 3.5 Precision results of 4 multiple executions of our approach on the analyzed projetcs

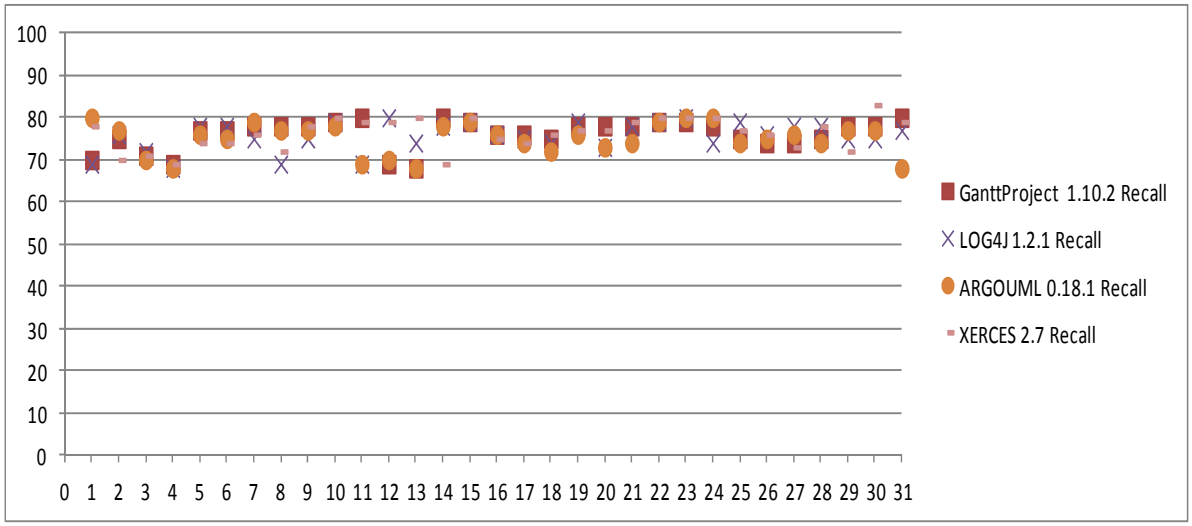


Figure 3.6 Recall results of multiple execution of our approach on the analyzed projects

To further analyze the effectiveness of our approach, we conducted a comparative study between the results of our approach and the results of two other approaches; i.e. the approach proposed by Moha et al. called DECOR (Moha et al., 2010) and our previous work presented in (Ghannem et al., 2011). We chose these two approaches because they analyzed the same projects and they considered two of the design defects that we are targeting in our experiments, namely the Blob and Functional Decomposition (FD). We also considered these two approaches because one of them relies on a search-based approach while the other does not. Indeed, DÉCOR (Moha et al., 2010) relies on the description of design defects to generate algorithms that detect these defects, while the approach in (Ghannem et al., 2011) uses a search based approach to automatically generate detection rules based on defects examples. Table 3.5 summarizes the results of our approach and the two other approaches in terms of precision for each of the Blob and FD defects. The DC (Data Class) defect was not considered by the other two approaches. We noticed that, in general, our proposal has better precision values than those given by the two approaches for the 4 analyzed projects. Overall the three approaches have very good results for the Blob defect. However, for the FD defect, the precision of our approach is quite high compared to DECOR's precision: the gap reaches nearly double for Log4J and Xerces projects (i.e., 100% vs. 54.5% and 90% vs. 51.7% respectively) and more than three times for GanttProjet (88% vs. 26.7%). On the other hand,

our previous work (Ghannem et al., 2011) which is search-based is more competitive to our current approach. These results can be explained by the fact that the FD defect is very difficult to describe in terms of rules and metrics, and a by-example approach is much more effective in this case.

Table 3.5 Comparison of the detection precision results of our approach to two other approaches

Project	**Design defect**	**Precision of our approach**	**Precision of (Ghannem et al., 2011)**	**Precision of DÉCOR (Moha et al., 2010)**
GanttProject v1.10.2	Blob	100 %	100 %	90 %
	FD	88 %	88 %	26.7 %
LOG4J v1.2.1	Blob	100 %	66 %	100 %
	FD	100%	82 %	54.5 %
ArgoUML v0.18.1	Blob	100 %	93%	86.2 %
	FD	86 %	82%	59.5 %
Xerces v2.7	Blob	100 %	97 %	88.6 %
	FD	90 %	88 %	51.7 %

To answer our second research question, we compared the results of our approach when applied to each of the three design defects we considered in this paper. Table 3.6 displays the average number of detected defects for each project and each defect; i.e., the values in Table 3.6 correspond to the average value of the recall per defect and project. We can notice that overall the majority of expected design defects are detected by our approach regardless of the defect type. For example, the recall varies from 86% to 100% for the Blob defect. Even in the case of the FD defect, the recall varies from 64% to 78% with an average of 70%. Accordingly, we can conclude that our approach is able to detect design defects regardless of their type. It's worth pointing out that, as for all example-based approaches, the results of our technique depend largely on the quality of the base of examples.

Table 3.6 Detected defects by our approach for each of the studied defects

Project	Average number of detected blobs	Average number of detected FD	Average number of detected DC
GanttProject v1.10.2	90% (9/10)	70% (12/17)	80% (8/10)
LOG4J v1.2.1	100% (3/3)	64% (7/11)	80% (4/5)
ArgoUML v0.18.1	86% (25/29)	78% (29/37)	88% (36/41)
Xerces v2.7	91% (40/44)	69% (20/29)	86% (50/58)

3.5 Related work

The proposal in this paper is related to work on detecting design defects in existing software. Existing work could be classified into two broad categories: non-search based techniques and search-based techniques for detecting design defects. Most of the approaches in the first category are based on rules specification. Erni et al. (Erni and Lewerentz, 1996) use metrics to evaluate frameworks with the goal of improving them. They introduce the concept of multi-metrics, n-tuples of metrics expressing a quality criterion (e.g., modularity). Marinescu (Marinescu, 2004) defined a list of rules relying on metrics to detect what he calls design flaws of OO design at method, class and subsystem levels. The main limitation of these approaches is the difficulty to manually define threshold values for metrics in the rules. To circumvent this problem, Alikacem et al. (Alikacem and Sahraoui, 2006) express defect detection as fuzzy rules, with fuzzy labels for metrics, e.g., small, medium, large. When evaluating the rules, actual metric values are mapped to truth values for the labels by means of membership functions. Although no thresholds need to be defined, still, it is not obvious to determine the membership functions. Moha et al. (Moha et al., 2010), in their DECOR approach, start by describing defect symptoms using an abstract rule language. These descriptions involve different notions, such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions which results in an important rate of false positives. Khomh et al. (Khomh et al., 2009) extended DECOR to support uncertainty and to sort the defect candidates accordingly. Uncertainty is managed by Bayesian belief networks that

implement the detection rules of DECOR. The detection outputs are probabilities that a class is an occurrence of a defect type.

Our approach is inspired by the approaches in the second category of work which use search-based techniques to suggest refactorings (e.g., (Harman and Tratt, 2007; Jensen and Cheng, 2010; Kessentini et al., 2008; O'Keeffe, 2008; Seng et al., 2006)). In these approaches design defects are not detected explicitly as the focus is put on detecting elements to change to improve the global quality. For example, a heuristic-based approach is presented in (Harman and Tratt, 2007; O'Keeffe, 2008; Seng et al., 2006) in which various software metrics are used as indicators for the need of a certain refactoring. In (Seng et al., 2006), a genetic algorithm is used to suggest refactorings to improve the class structure of a system. The algorithm uses a fitness function that relies on a set of existing object oriented metrics. Harman and Tratt (Harman and Tratt, 2007) propose to use the Pareto optimality concept to improve search-based refactoring approaches when the evaluation function is based on a weighted sum of metrics. Both the approaches in (Seng et al., 2006) and (Harman and Tratt, 2007) were limited to the Move Method refactoring operation. In (O'Keeffe, 2008), the authors present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics. The results of the experiments on five open-source systems showed that hill-climbing performs better than the other algorithms. In (Kessentini et al., 2010), Kessentini et al proposed an approach, based on search-based techniques, for the automatic detection of potential design defects in code. The detection is based on the notion that the more code deviates from good practices, the more likely it is bad. In both (Ghannem et al., 2011) and (Ouni et al., 2013), a search-based approach is used to generate rules that detect design defects in existing code. Contrary to these two approaches, our current proposal does not generate detection rules; it uses defect examples to identify potential defects. Also the results from our experiments proved that our current proposal yields better results than our previous work.

In our approach, we tackled the defects detection problem at the model level specifically in class diagrams. We circumvent the above-mentioned problems related to the use of rules, metrics, symptoms and the manual adaptation/calibration effort by identifying directly the defect based on defects examples.

3.6 Conclusion

In this paper, we presented a novel search-based approach to improve the automation of design defects detection. We proposed an algorithm that is an adaptation of Genetic Algorithms (GA) to exploit an existing corpus of known design defects and detect design defects in class diagrams. The proposed fitness function aims to maximize: 1) the structural similarity between the model under analysis (i.e., class diagram) and the models in the base of examples and, 2) the number of detected defects. We tested the approach on four open source projects targeting the detection of three design defects. The results of our experiment have shown that the approach is stable regarding its correctness and completeness. The approach has also significantly increased the average precision and recall when compared to other approaches.

As part of future work, we plan first to cover all design defects potentially detectable in class diagrams. We plan also to extend our base of examples with additional badly-designed models in order to take into consideration more programming contexts. We also want to study and analyze the impact of using domain-specific examples on the effectiveness of the approach. Actually, we kept the random aspect that characterizes genetic algorithms even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to detect defects in the one under analysis. Finally, we want to apply the approach on other open source projects and further analyze the type of defects that are correctly detected when using examples.

CHAPTER 4

MODEL REFACTORING USING EXAMPLES: A SEARCH BASED APPROACH

Adnane Ghannem, Ghizlane El Boussaidi[1] and Marouane Kessentini[2]

[1]Department of Software and IT Engineering, École de Technologie Supérieure,

1100 Notre-Dame West, Montreal, Quebec, (H3C 1K3) Canada

[2]Department of Computer and Information Science, University of Michigan - Dearborn

4901 Evergreen Road, Dearborn, MI 48128 USA

This paper has been published in Journal of Software: Evolution

and Process

ABSTRACT

One of the important challenges in model-driven engineering is how to improve the quality of the models' design in order to help designers understanding them. Refactoring represents an efficient technique to improve the quality of a design while preserving its behavior. Most of existing work on model refactoring relies on declarative rules to detect refactoring opportunities and to apply the appropriate refactorings. However, a complete specification of refactoring opportunities requires a huge number of rules. In this paper, we consider the refactoring mechanism as a combinatorial optimization problem where the goal is to find good refactoring suggestions starting from a small set of refactoring examples applied to similar contexts. Our approach, named MOdel REfactoring by eXample (MOREX), takes as input an initial model to refactor, a set of structural metrics calculated on both initial model and models in the base of examples, and a base of refactoring examples extracted from different software systems and generates as output a sequence of refactorings. A solution is defined as a combination of refactoring operations that should maximize as much as possible the structural similarity based on metrics between the initial model and the models in the base of examples. A heuristic method is used to explore the space of possible refactoring solutions. To this end, we used and adapted a genetic algorithm (GA) as a global heuristic search. The validation results on different systems of real-world models taken from open source projects confirm the effectiveness of our approach.

Keywords: Software maintenance, Model evolution, Model refactoring, Refactoring by example, Heuristic method, and Genetic algorithm.

4.1 Introduction

To cope with the changing and growing business needs, software systems are constantly evolving. Software evolution activities can span from maintenance to an entire replacement of the system (Seacord et al., 2003). Software maintenance is considered the most expensive activity in the software system lifecycle (Lientz et al., 1978). According to the ISO/IEC 14764 standard, the maintenance process includes the necessary tasks to modify existing software while preserving its integrity (ISO/IEC, 2006). Maintenance tasks can be seen as incremental modifications to a software system that aim to add or adjust some functionality or to correct some design flaws and fix some bugs. However, as the time goes by, the system's conceptual integrity erodes (Seacord et al., 2003) and its quality degrades; this deterioration is known in the literature as the software decay problem (Fowler, 1999). Therefore, maintenance tasks become more complex and costly.

A common and widely used technique to cope with this problem is to continuously restructure the software system to improve its structure and design. The process of restructuring object oriented systems is commonly called refactoring (Mens and Tourwé, 2004). According to Fowler (Fowler, 1999), refactoring is the disciplined process of cleaning up code to improve the software structure while preserving its external behavior. Automating refactoring operations necessarily helps coping with software complexity and keeping the maintenance costs from increasing. Many researchers have been working on providing support for refactoring operations (e.g., (Opdyke, 1992), (Fowler, 1999), and (Moha, 2008)). Existing tools provide different environments to manually or automatically apply refactoring operations to correct, for example, code smells (Du Bois et al., 2004). Indeed, existing work has, for the most part, focused on refactorings at the source code level. Very few approaches tackled the refactoring process at the model level (e.g., (El-Boussaidi and Mili, 2011), (Mens et al., 2007a) and (Zhang et al., 2005)). Nevertheless, models are primary artifacts within the model-driven engineering (MDE) approach which has emerged as a promising approach to manage software systems' complexity and specify domain concepts effectively (Douglas, 2006). In MDE, abstract models are refined and successively transformed into more concrete

models including executable source code. Evolution of models and the transformations that manipulate them is crucial to MDE approaches however the maintenance process is still focused on source code.

Actually, the rise of MDE increased the interest and the needs for tools supporting refactoring at the model-level. Indeed, such a tool may be of great value for novice designers as well as experienced ones when refactoring existing models. However there are many open and challenging issues that we must address when building such a tool. Mens and Tourwé (Mens et al., 2007a) argue that most of the refactoring tools offer a semi-automatic support because part of the necessary knowledge for performing the refactoring remains implicit in designers' heads. Indeed, recognizing opportunities of model refactoring remains a challenging issue that is related to the model marking process within the context of MDE which is a notoriously difficult problem that requires design knowledge and expertise (El-Boussaidi and Mili, 2008). Finding refactoring opportunities in source code has relied, for the most part, on quality metrics (e.g., (Moha et al., 2010), (Munro, 2005), (Marinescu, 2004)). However, some of these metrics (e.g., number of lines of code) and refactorings (e.g., removing duplicate code) do not apply at the model-level. Hence the designer needs to identify the useful and applicable metrics for a given model of the system and decide how to correctly combine these metrics to detect and propose a refactoring. In addition, existing work on refactoring relies on declarative rules to detect and correct defects (i.e., refactoring opportunities) and the number of types of these defects can be very large (Kessentini et al., 2011b). This problem's complexity is strongly increased when the designer is looking for an appropriate sequence of refactorings that corrects the entire set of the system's defects.

In this paper, we hypothesize that the knowledge required to propose appropriate refactorings for a given object-oriented model may be inferred from other existing models' refactorings when there is some similarities between these models and the given model. We propose MOREX (MOdel REfactoring by eXample), an approach to automate model refactoring using heuristic based search. MOREX relies on a set of refactoring examples to propose sequences of refactorings that can be applied on a given object-oriented model. The

refactoring is seen as an optimization problem where different sequences of refactorings are evaluated depending on the similarity between the model under analysis and the refactored models in the examples at hand. Our approach takes as input an initial model which we want to refactor, a base of examples of refactored models and a list of metrics calculated on both the initial model and the models in the base of examples, and it generates as output a solution to the refactoring problem. In this case, a solution is defined as a sequence of refactoring operations that should maximize as much as possible the similarity between the initial model and the models in the base of examples. Due to the very large number of possible solutions (i.e., refactoring combinations), a heuristic method is used instead of an enumerative one to explore the space of possible solutions. Since the search space is very large, we use and adapt a genetic algorithm as a global heuristic search.

The primary contributions of the paper can be summarised as follows:

1. We introduce a new refactoring approach based on the use of examples. Our proposal does not require the user to define explicitly defect types, but only to have some refactoring examples; it does not require an expert to write detection or correction rules manually; and it combines detection and correction steps.

2. We report the results of an evaluation of our approach; we used refactoring examples extracted from eight object-oriented open source projects. We applied an eight-fold cross validation procedure. For each fold, one open source project is evaluated by using the remaining seven systems as bases of examples. The average values of precision and recall computed from 31 executions on each project are around 85% which allows us to say that the obtained results are promising. The effectiveness of our approach is also assessed using a comparative study between our approach and two other approaches.

The paper is organized as follows. Section 4.2 is dedicated to the basic concepts. Section 4.3 presents the overall approach and the details of our adaptation of the genetic algorithm to the

model refactoring problem. Section 4.4 describes the implementation and the experimental setting. Section 4.5 presents and discusses the experimental results. Related works are discussed in section 4.6 and we conclude and outline some future directions to our work in section 4.7.

This section defines some relevant concepts to our proposal, including model refactorings, software metrics and heuristic search.

4.2 Basic concepts

This section defines some relevant concepts to our proposal, including model refactorings, software metrics and heuristic search.

4.2.1 Model refactorings

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." (Fowler and Beck, 1999). Model refactoring is a controlled technique for improving the design (e.g., class diagrams) of an existing model. It involves applying a series of small refactoring operations to improve the model quality while preserving its behavior. Many refactorings were proposed and codified in the literature (see e.g., (Fowler, 1999)). In our approach, we considered a subset of the 72 refactorings defined in (Fowler, 1999); we considered only those refactorings that can be applied to class diagrams as an example of design models. Indeed, some of the refactorings in (Fowler, 1999) may be applied on design models (e.g. Move_Method, Rename_method, Move_Field, Extract_Class etc.) while others cannot be (e.g. Extract Method, Inline Method, Replace Temp With Query etc.). The refactoring configuration for the experiments of our approach reported here consisted of the twelve (12) refactorings described below (see Table 4.1). The choice of these refactorings was mainly based on two factors: 1) they apply at the model-level (i.e., we focused on class diagrams); 2)

they can be linked to a set of model metrics (i.e. metrics which are impacted when applying these refactorings). The considered metrics are presented in the following subsection.

Table 4.1 Considered refactorings in the MOREX approach

Refactoring Name	Description
Extract class	Create a new class and move the relevant fields and methods from the old class into the new class
Rename method	Rename method with a name that reveals its purpose. This refactoring is intended to give more comprehensiveness to the model design.
Push down method	Move behavior from a superclass to a specific subclass, usually because it makes sense only there.
Push down field	Move a field from super class to a specific subclass, usually because it makes sense only there.
Rename parameter	Rename a parameter within the method parameter list.
Add parameter	Add a new parameter to the method parameter list.
Move field	Move a field from a source class to the class destination when it's more used by the second one than the class on which it is defined.
Move method	Move a method from a class to another one when it's using or used by more features of the destination class than the class on which it is defined.
Pull up method	Move a method from some class(es) to the immediate superclass. This refactoring is intended to help eliminate duplicate methods among sibling classes, and hence reduce code duplication in general.
Pull up field	Move a field from some class(es) to the immediate superclass. This refactoring is intended to help eliminate duplicate field declarations in sibling classes.
Extract interface	Create an interface class when many classes use the same subset of a class's interface, or two classes have part of their interfaces in common.
Replace inheritance with delegation	Change the inheritance relation by a delegation when the subclass uses only part of a super classes interface or does not want to inherit data.

4.2.2 Quality Metrics

Quality metrics provide useful information that help assessing the level of conformance of a software system to a desired quality such as *evolvability* and *reusability* (Fenton and

Pfleeger, 1998). Metrics can also help detecting some similarities between software systems. The most widely used metrics for class diagrams are the ones defined by Genero et al. (Genero et al., 2002). In the context of our approach, we used the eleven (11) metrics defined in (Genero et al., 2002) to which we have added a set of simple metrics (e.g., number of private methods in a class, number of public methods in a class) that we have defined for our needs. The metrics configuration for the experiments reported here consisted of the sixteen quality metrics described below in Table 4.2. All this metrics are related to the class entity which is the main entity in a class diagram. Some of these metrics represent statistical information (e.g. number of methods, attributes, etc.) and others give information about the position of the class through its relationships with the other classes of the model (e.g. number of associations). All these metrics have a strong link with the refactorings presented in the previous section.

Table 4.2 Considered metrics in the MOREX approach

Metric name	Description
Number of attributes(**NA**)	The total number of attributes of a given class.
Number of private attributes(**NPvA**)	The total number of private attributes of a given class.
Number of public attributes(**NPbA**)	The total number of public attributes of a given class.
Number of protected attributes(**NProtA**)	The total number of protected attributes of a given class.
Number of methods(**NMeth**)	The total number of methods of a given class.
Number of private methods (**NPvMeth**)	The total number of private methods in a given class.
Number of public methods (**NPbMeth**)	The total number of public methods in a given class.
Number of protected methods (**NProtMeth**)	The total number of protected methods in a given class.
Number of associations (**NAss**)	The total number of associations.
Number of aggregations (**NAgg**)	The total number of aggregation relationships.
Number of dependencies (**NDep**)	The total number of dependency relationships.
Number of generalizations (**NGen**)	The total number of generalisation relationships (each parent-child pair in a generalization relationship).
Number of aggregations hierarchies	The total number of aggregation hierarchies.

(**NAggH**)	
Number of generalization hierarchies (**NGenH**)	The total number of generalisation hierarchies.
DIT (**DIT**)	The DIT value for a class within a generalisation hierarchy is the longest path from the class to the root of the hierarchy.
HAgg (**HAgg**)	The HAgg value for a class within an aggregation hierarchy is the longest path from the class to the leaves.

4.2.3 Heuristic search

Heuristic search enables to promote discovery or learning (Pearl, 1984). It consists to search a space of possible solutions to a problem, or to find an acceptable approximate solution, when an exact algorithmic method is unavailable or too time-consuming (e.g. complex combinatorial problems). There are a variety of methods which perform heuristic search as hill climbing (Mitchell, 1998), simulated annealing (Kirkpatrick et al., 1983), genetic algorithms (Goldberg, 1989), etc. In this section we give an overview of genetic algorithms (GA) and we describe how a GA can be used to generate sequences of refactorings. GA is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution (Koza, 1992). The basic idea behind GA is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem. In GA, a solution can be represented as a vector. Each individual (i.e. a solution) of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. Exploration of the search space is achieved by selecting individuals (in the current population) that have the best fitness values and evolving them by using of genetic operators, such as crossover and mutation. The crossover operator insures generation of new children, or offspring, based on parent individuals. The crossover operator allows transmission of the features of the best fitted parent individuals to new individuals. Each pair of parent individuals produces two children (new solutions). Finally, mutation operator is applied to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping local optima found during the search. Mutation is often performed with a low

probability in GAs (Goldberg, 1989). Once selection, mutation and crossover have been applied according to given probabilities, individuals of the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. This criterion usually corresponds to a fixed number of generations. The result of GA (the best solution found) is the fittest individual produced along all generations.

Hence to apply GA to a specific problem (i.e., the refactoring problem in our context), the following elements have to be adapted to the problem at hand:

1. Representation of the individuals,
2. Creation of a population (i.e. a generation) of individuals,
3. Evaluation of individuals using a fitness function,
4. Selection of the (best) individuals to transmit from one generation to another,
5. Creation of new individuals using genetic operators (crossover and mutation) to explore the search space,
6. Generation of a new population using the selected individuals and the newly created individuals.

4.3 A heuristic search approach to model refactoring

4.3.1 Overview of the Approach

The approach proposed in this paper exploits examples of model refactorings and a heuristic search technique to automatically suggest sequences of refactorings that can be applied on a given model. The general structure of our approach is illustrated by Figure 4.1.

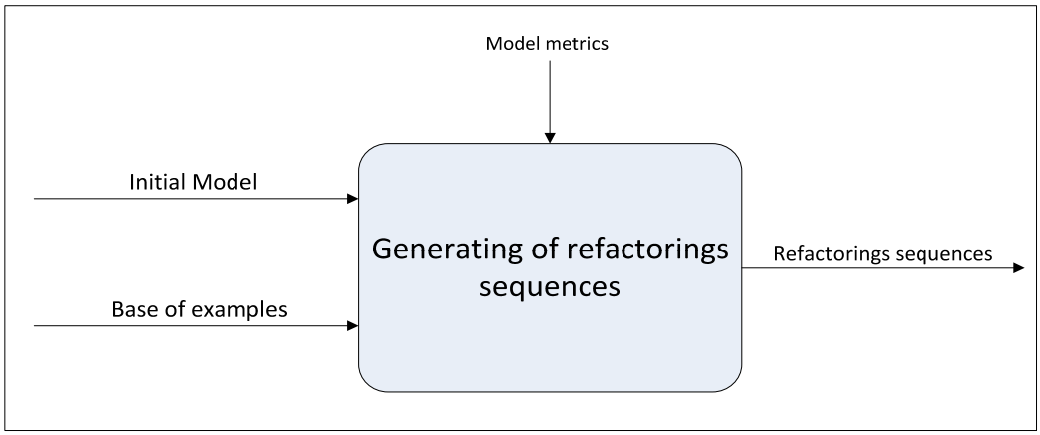


Figure 4.1 Approach overview

Our refactoring approach takes as inputs an initial model and a set of models in the base of examples and their related refactorings, and takes as controlling parameters a set of quality metrics. The approach generates a set of refactoring operations that represents refactoring opportunities for the initial model. The process of generating a sequence of refactorings (Figure 4.2) can be viewed as the mechanism that finds the best way to combine refactoring operations among the list proposed in the models in the base of examples, in such a way to best maximize the similarity between entities to be refactored in the initial model and entities of the models in the base of examples that have undergone the refactoring operations composing the sequence.

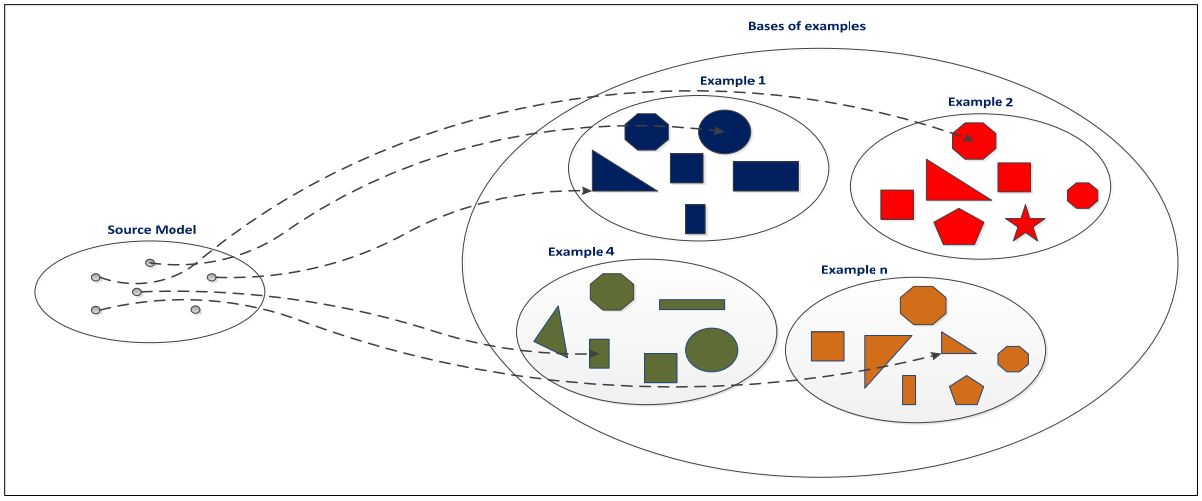


Figure 4.2 Illustration of proposed generation process

Accordingly the algorithm that generates relevant sequences of refactorings has to explore a huge search space. In fact, the search space is determined by the number of possible refactoring combinations. Formally, if m is the number of available refactoring operations, then the number R of possible refactoring subsets is equal to $R = 2^m$. If c is the cardinality of a subset of possible refactorings to which we add the order, then the number of permutations will equal to c!. In this context, the number NR of possible combinations that have to be explored by the algorithm is given by:

$$NR = \sum_{i=1}^{2^m} c_i! \tag{4.1}$$

But this brute force method is infeasible in practice, due to the expensive computation. Even for a small number of refactorings (for m = 5, NR is 3840), the NR value quickly becomes larger, since the same refactoring operations can be applied several times on different parts of the model (e.g., class, method, attribute). Due to this large number of possible refactoring solutions, we resorted to a heuristic-based optimization method to solve the problem. Hence we considered the model refactorings' generation as an optimization problem, and we adapted the genetic algorithm (Koza, 1992) to this problem in order to find an optimal solution (i.e., a sequence of refactorings) that maximizes the similarity between the entities (class, methods, attributes) of the initial model and those of the models in the base of examples.

4.3.2 Adaptation of the genetic algorithm to model refactoring

A high-level view of our adaptation of GA to the model refactoring problem is given in Algorithm 4.1. As this figure shows, the algorithm takes as input a set of quality metrics and a set of model refactoring examples.

```
Input: Set of quality metrics
Input: Set of model refactoring examples
Output: A sequence of refactorings

1     I:= set_of(CIM, CBE, set of the refactorings applied to CBE that are applicable to CIM)
2     P:= set_of(I)
3     initial_population(P, Max_size)
4     repeat
5           for all I P do
7                 fitness(I) := Sum [similarity (CIM, CBE) for all (CIM, CBE) belonging to I]
8           end for
9           best_solution := best_fitness(I);
10          P := generate_new_population(P)
11:         it:=it+1;
12:   until it=max_it or fitness(best_solution) = 0;
13:   return best_solution
```

Algorithm 4.1 High level pseudo code for GA adaptation to our problem

Lines 1–3 construct an initial GA population, which is a set of individuals that stand for possible solutions representing sequences of refactorings that can be applied to the classes of the initial model. An individual is a set of triplets; a triplet is called a block and it contains a class of the initial model denoted as CIM, a class of the base of examples denoted as CBE, and a set of refactorings that were applied to CBE and that are applicable to CIM. To generate an initial population, we start by defining the maximum individual size in terms of a maximum number of blocks composing an individual. This parameter can be specified either by the user or randomly. Thus, the individuals have different sizes. Then, for each individual, the blocks are randomly built; i.e., a block is composed by:

1. A pair (CIM, CBE) of randomly matched classes; i.e., one class (CIM) from the initial model that is under analysis and its randomly matched class (CBE) from the base of examples.
2. A set of refactorings that we can possibly apply on the class CIM from the initial model extracted from the set of refactorings that were applied to its matched class CBE from the base of examples.

Individuals' representation is explained in more detail in section 4.3.3.

Lines 4–13 encode the main GA loop, which explores the search space and constructs new individuals by changing the matched pairs (CIM, CBE) in blocks. During each iteration, we evaluate the quality of each individual in the population. To do so, we use a fitness function that sums the similarities between the classes CMI and CBE of each block composing the individual (line 7). Computation of the fitness function of an individual is described in more detail in section 4.3.5. Then we save the individual having the best fitness (line 9). In line 10, we generate a new population (p+1) of individuals from the current population by selecting 50% of the best fitted individuals from population p and generating the other 50% of the new population by applying the crossover operator to the selected individuals; i.e., each pair of selected individuals, called parents, produces two children (new solutions). Then we apply the mutation operator, with a probability, for both parents and children to ensure the solution diversity; this produces the population for the next generation. The mutation probability specifies how often parts of an individual will mutate. Selection, crossover and mutation are described in details in section 4.3.4.

The algorithm stops when the termination criterion is met (Line 12) and returns the best solution found during all iterations (Line 13). The termination criteria can be a maximum number of iteration or the best fitness function value. However, the best fitness function value is difficult to predict and sometimes it takes very long time to converge towards this value. Hence, our algorithm is set to stop when it reaches the maximum iteration number or the best fitness function value.

In the following subsections, we describe in details our adaption of GA to the model refactoring problem. To illustrate this adaption, we use an example of a class diagram as a model to refactor. Thus, the base of examples is a set of refactorings' examples on class diagrams.

4.3.3 Individual Representation

An individual is a set of blocks. A block contains three parts as shown by Figure 4.3: the first part contains the class CIM chosen from the initial model (model under analysis), the second part contains the class CBE from the base of examples that was matched to CIM, and finally the third part contains a list of refactorings which is a subset of the refactorings that were applied to CBE (in its subsequent versions) and that can be applied to CIM.

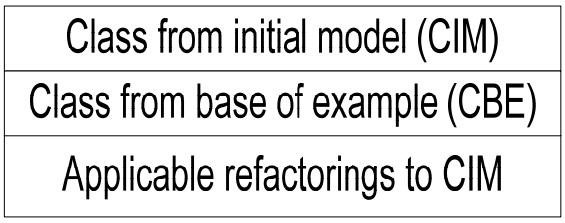


Figure 4.3 Block representation

In our approach, we represented models using predicates. However, we used a slightly different predicate format for representing the classes of the model under analysis and those in the base of examples. Figure 4.4 illustrates the predicate format used to represent a class (CIM) from the initial model while Figure 4.5 illustrates the predicate format to represent a class (CBE) from the base of examples. The representation of a CBE class includes a list of refactorings that were applied to this class in a subsequent version of the system's model to which CBE belongs. The subset of a CBE subsequent refactorings that are applicable to a CIM class constitutes the third part of the block having CIM as its first part and CBE as its second part. Hence, the selection of the refactorings to be considered in a block is conformed to some constraints to avoid conflicts and incoherence errors. For example, if we have a

Move_attribute refactoring operation in the CBE class and the CIM class doesn't contain any attribute, then this refactoring operation is discarded as we cannot apply it to the CIM class.

```
Class (Class Name; visibility)
{
        Attribute(Attribute Name; Type; visibility; ReturnType)
        ...
        Method(Method Name; [Parameters]; Visibility; Return Type;)
        ...
        Relation(Name of the source class; Name of the destination class;  Type of the relation)
        ...
}
```

Figure 4.4 Class representation in the initial model

```
Class (Class Name; visibility)
{
        Attribute(Attribute Name; Type; visibility; ReturnType)
        ...
        Method(Method Name; [Parameters]; Visibility; Return Type;)
        ...
        Relation(Name of the source class; Name of the destination class;  Type of the relation)
        ...
        Refactoring(Refactoring Name (Parameters))
        ...
}
```

Figure 4.5 Class representation in the base of examples

The bottom part of Figure 4.6 shows an example of an individual (i.e., a candidate solution) that we extracted from our experiment described in section 4.4. This individual is composed of several blocks. The first block (encircled in Figure 4.6) was produced by matching a class from the model under analysis (ResourceTreeTable) and a class from the base of example (mxLayoutManager) shown in the top part of Figure 4.6. Class mxLayoutManager has undergone two refactorings which can be applied to class ResourceTreeTable. Hence, in this context, the two refactorings are included in the refactoring sequence that constitutes the third part of the first block. It's important to highlight that a class from the initial model can be included only in a single block of a given individual. The top part of Figure 4.7 shows another example of an individual. Each block of this individual contains one refactoring

operation. The bottom part of Figure 4.7 shows the fragments of an initial model before and after the sequence of refactorings proposed by the individual (at the top of the figure) were applied. Hence the individual represents a sequence of refactoring operations to apply and the classes of the initial model on which they apply.

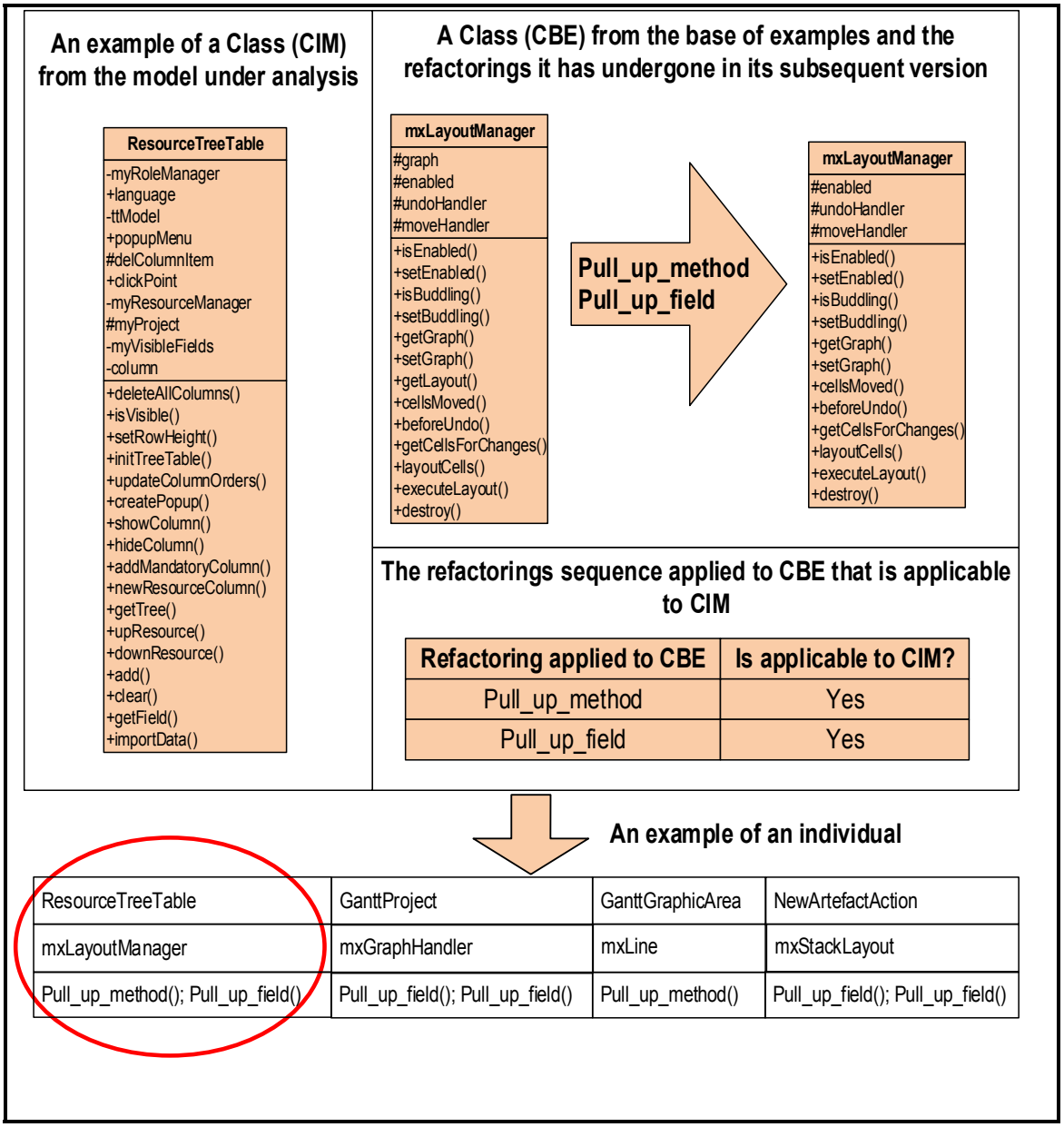


Figure 4.6 Example extracted from our experiment

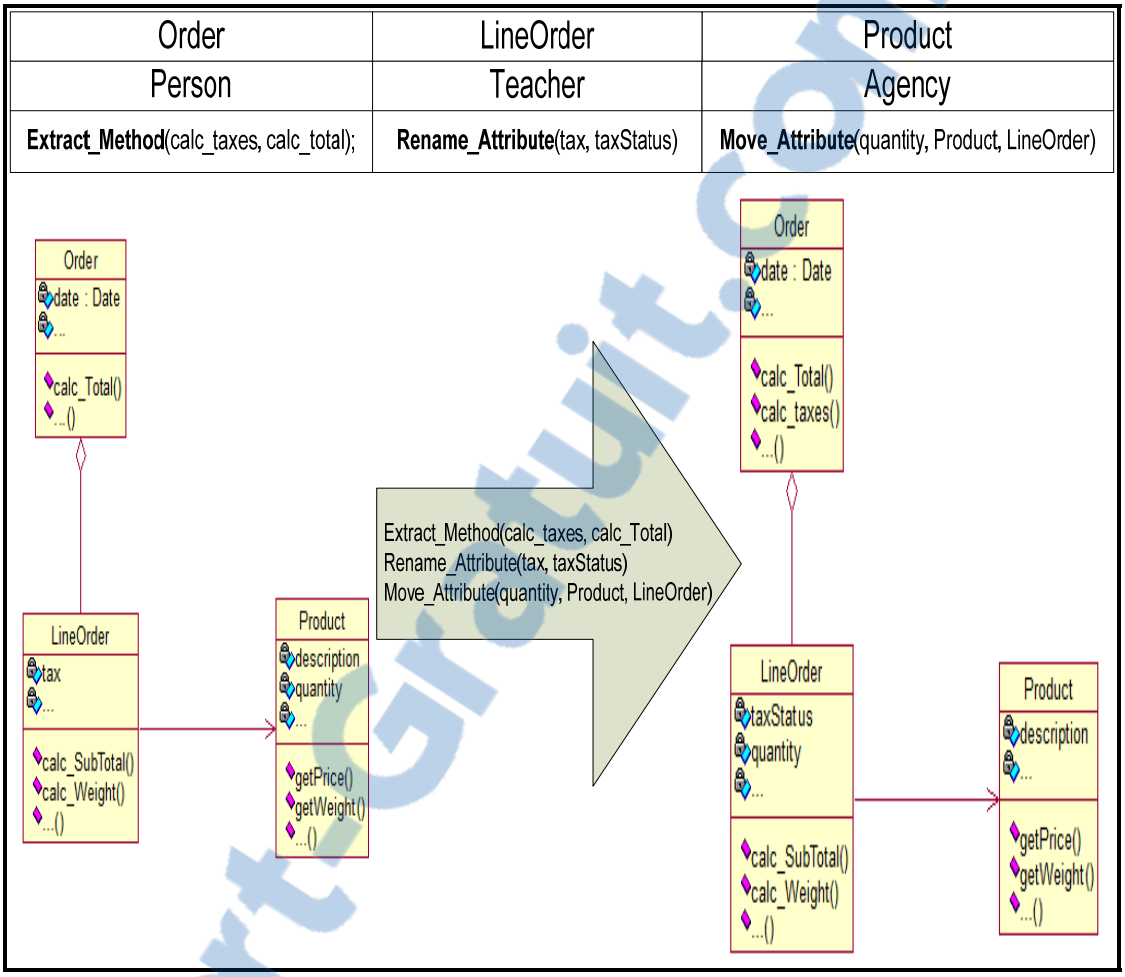


Figure 4.7 An individual as a sequence of refactorings

4.3.4 Genetic Operators

Selection

We used the stochastic universal sampling (SUS) (Koza, 1992) to select individuals that will undergo the crossover and mutation operators to produce a new population from the current one. In the SUS, the probability of selecting an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select 50% of individuals from population p for the new population p+1. These (population_size/2) selected individuals

will be transmitted from the current generation to the new generation and they will "give birth" to another (population_size/2) new individuals using crossover operator.

Crossover

For each crossover, two individuals are selected by applying the *SUS* selection (Koza, 1992). Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring P'_1 and P'_2 from the two selected parents P_1 and P_2. It is defined as follows: A random position, k, is selected. The first k refactorings of P_1 become the first k elements of P'_2. Similarly, the first k refactorings of P_2 become the first k refactorings of P'_1. The rest of refactorings (from position k+1 until the end of the sequence) in each parent P_1 and P_2 are kept. For instance, Figure 4.8 illustrates the crossover operator applied to two individuals (parents) P_1 and P_2. The position k takes the value *2*. The first two refactorings of P_1 become the first two elements of P'_2. Similarly, the first two refactorings of P_2 become the first k refactorings of P'_1.

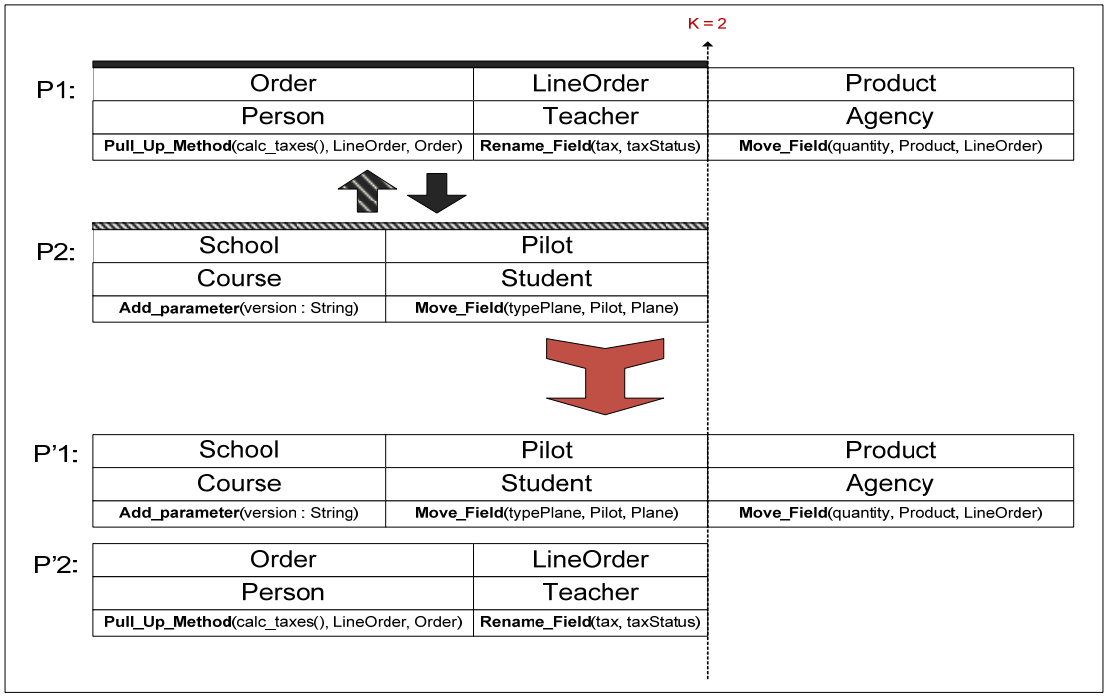


Figure 4.8 Crossover operator

Mutation

The mutation operator consists of randomly changing one or more dimensions (block) in the solution (vector). Hence, given a selected individual, the mutation operator first randomly selects some blocks in the vector representation of the individual. Then the CBE of the selected block is replaced by another CBE chosen randomly from the base of examples.

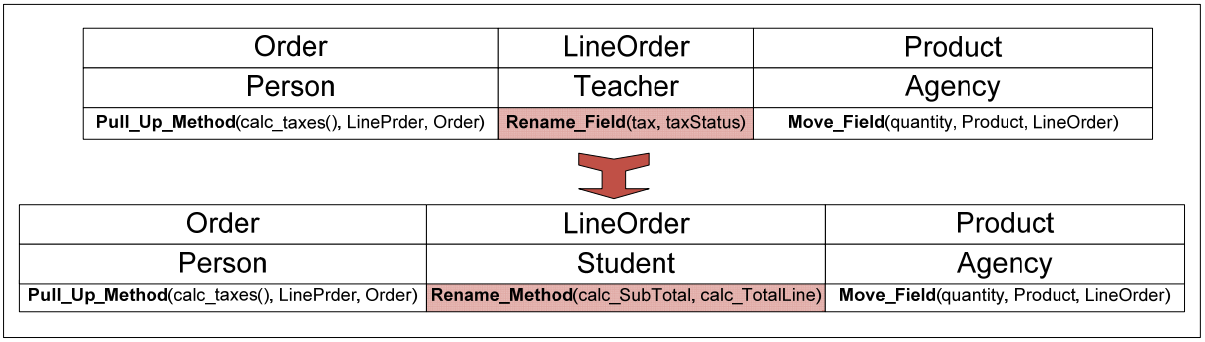


Figure 4.9 Mutation operator

Figure 4.9 illustrates the effect of a mutation that replaced the refactoring *Rename_Attribute (tax, taxStatus)* applied to the class *LineOrder* (initial model) which is extracted from the class *Teacher* (base of examples) by the refactoring *Rename_Method(calc_SubTotal, calc_TotalLine)* extracted from the new matched class *Student* (base of examples) and applied to the class *LineOrder* (initial model).

4.3.5 Decoding of an Individual

The quality of an individual is proportional to the quality of the refactoring operations composing it. In fact, the straight way to evaluate the quality of an individual is to apply its sequence of refactorings to the model under analysis. However, our goal is to find a way to infer correct refactorings using the knowledge that has been accumulated through refactorings of other models of past projects. Specifically, we want to exploit the similarities between the actual model and other models to infer the sequence of refactorings that we must

apply. Our intuition is that a candidate solution that displays a high similarity between the classes of the model and those chosen from the example base should give the best sequence of refactorings.

Practically, the evaluation of an individual should be formalized as a mathematical function called “fitness function”. The goal is to define an efficient and simple (in the sense not computationally expensive) fitness function in order to reduce the computational complexity. As discussed above, the fitness function aims to maximize the similarity between the classes of the model in comparison to the ones in the base of examples. In this context, we define the fitness function of a solution as:

$$f = \sum_{j=1}^{n} Similarity(CIM, CBE) = \sum_{j=1}^{n} \sum_{i=1}^{m} |CIM_i - CBE_i| \quad (4.2)$$

where n and m are respectively the number of blocks in the solution and the number of metrics considered in this project. CIM and CBE are respectively the class from the initial model and the class from the base of examples that belong to the j^{th} block. CIM_i is the i^{th} metric value of the class CIM while CBE_i is the i^{th} metric value of the class CBE. Figure 4.10 illustrates the way we compute the similarity between two given classes using their metrics’ values.

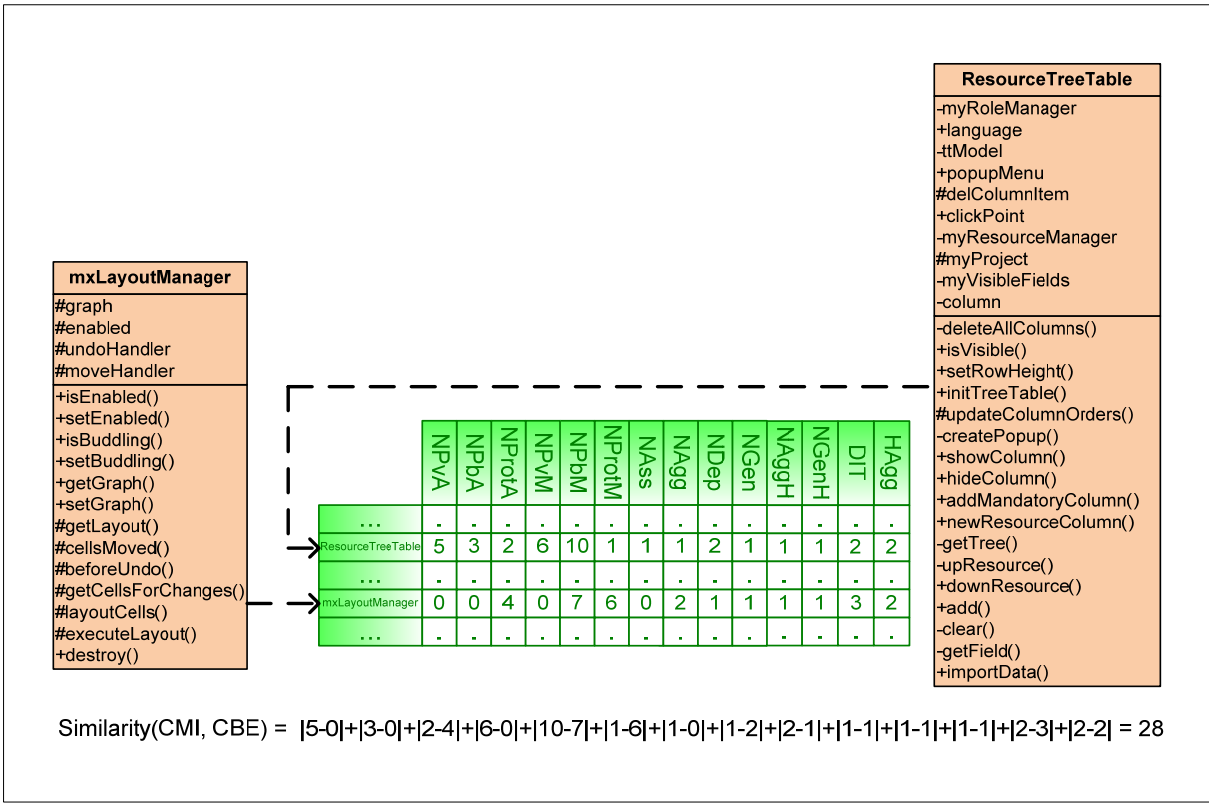


Figure 4.10 Computing the similarity between two classes

To illustrate how the fitness function is computed, we consider a system containing three classes as shown in Table 4.3 and a base of examples containing three classes shown in Table 4.4. In this example we use six metrics and these metrics are given for each class in the model in Table 4.3 and each class of the base of examples in Table 4.4.

Table 4.3 Classes from the initial model and their metrics values

Class in the initial model	***NPvA***	***NPbA***	***NPbMeth***	***NPvMeth***	***NAss***	***NGen***
Order	3	0	5	2	2	1
LineOrder	4	1	3	1	1	1
Product	2	2	6	0	1	0

Table 4.4 Classes from the base of examples and their metrics values

Class in the base of examples	***NPvA***	***NPbA***	***NPbMeth***	***NPvMeth***	***NAss***	***NGen***
Student	2	1	3	0	3	0
Agency	4	4	1	2	0	3
Plane	5	1	4	0	1	0

Consider the example of two individuals I_1 and I_2 respectively composed by one block (*Order* from the model and *Agency* from the BE) and two blocks (*LineOrder*/*Student* and *Product*/*Plane*). The fitness function calculated on these solutions has the value:

$$\boldsymbol{f_{I_1}} = |3-4| + |0-4| + \cdots + |1-3| = 13 \tag{4.3}$$

$$\boldsymbol{f_{I_2}} = (|4-2| + |1-1| + \cdots + |1-0|) + (|2-5| + |2-1| + \cdots + |0 - 0|) = 12 \tag{4.4}$$

If we consider a population composed by only these two individuals, the evaluation process chooses the one which has the minimum value of fitness function, then I_2 will be chosen as best individual.

4.4 Implementation and experimental settings

In this section, we describe our experimental setup. To set the parameters of GA for the search strategies, we performed several tests and the final parameters' values were set to a minimum of 1000 iterations for the stopping criterion, to 2 as the minimum length of a solution in terms of number of block, and to 25 as the maximum length of a solution. We also set the crossover probability to 0.9 and the mutation probability to 0.5. These values were obtained by trial and error. We selected a high mutation rate because it allows the continuous diversification of the population which discourage premature convergence to occur.

4.4.1 Supporting tool

To validate our approach, we implemented a parser which analyses Java source code and generates a predicate model as illustrated by Figure 4.4. We used this parser to generate predicate models from 8 Java open source projects. To build the base of examples, we completed the generated models by manually entering the refactoring operations extracted with Ref-Finder (Kim et al., 2010), that these projects have undergone. The Ref-Finder tool allows detection of complex refactorings (68 refactorings) between two program versions using logic-based rules executed by a logic programming engine. Ref-Finder helps finding refactorings that a system has undergone by comparing different versions of the system. We used the refactorings returned by Ref-finder for two reasons; to build the base of examples and to compute the precision and recall of our approach.

4.4.2 Research questions

The goal of our experiment is to evaluate the efficiency of our approach in generating relevant sequences of refactorings. In particular the experiment aimed at answering the following research questions:

RQ1: To what extent can the proposed approach generate the correct sequences of refactorings?
RQ2: Is the approach stable? This question aims to verify if the returned refactorings are correct for different executions of the approach.

To answer RQ1, we evaluated the precision and recall of our approach by applying it on a set of existing projects for which we had several versions and hence information about the refactorings they had undergone. To answer RQ2, we run GA multiple times (31 runs for each project) and observe the algorithm's behaviour in terms of precision and recall scores through these executions.

4.4.3 Selected projects for the analysis

To answer the research questions reported above, we used 8 open-source Java projects to perform our experiments. The projects are:

1. Ant (v1.8.4): A Java library that is mainly used for building Java applications. Ant provides support to compile, assemble, test and run Java applications.
2. GanttProject (v0.10): A Java project that supports project management and scheduling.
3. JabRef (v2.7): A graphical application for managing bibliographical databases.
4. JGraphx (v1.10.4.0): A Java Swing diagramming (graph visualisation) library.
5. JHotDraw (v5.2): A framework for the creation of drawing editors.
6. JRDF (v0.5.6.2): A Java library for parsing, storing and manipulating RDF(Resource Description Framework).
7. Xerces (v2.5): A set of parsers compatible with Extensible Markup Language (XML).
8. Xom (v1.2.8): A new XML object model.

We have chosen these open source projects because they are medium-sized open-source projects and most of them were analyzed in related work (e.g., (Moha et al., 2010), (Kim et al., 2010), (Ghannem et al., 2011) and (Ouni et al., 2013)). Most of these open source projects have been actively developed over the past 10 years. Table 4.5 provides some relevant information about these projects.

Table 4.5 Case study settings

Model	**Number of classes**	**Number of methods**	**Number of attributes**	**Number of expected refactoring operations**
Ant 1.8.4	824	2090	1048	139
GanttProject 2.0.10	479	960	495	91
JabRef 2.7	594	253	237	32
JGraphx 1.10.4.0	191	1284	420	96

JHotDraw 5.2	160	519	141	71
JRDF v0.5.6.2	734	19	10	41
Xerces 2.5	625	2113	1408	182
Xom 1.2.8	252	186	31	36

In our validation we use one project as the system under analysis and the other 7 projects as the base of examples. Then, we compare the refactoring sequences returned by the algorithm with the ones returned by Ref-finder when executed on the same version of the system under analysis and the following version.

4.4.4 Measures of precision and recall

To assess the accuracy of our approach, we compute the measures precision and recall originally stemming from the area of information retrieval. When applying precision and recall in the context of our study, the precision denotes the fraction of correctly detected refactoring operations among the set of all detected operations. The recall indicates the fraction of correctly detected refactoring operations among the set of all actually applied operations (i.e., how many operations have not been missed). In general, the precision denotes the correctness of the approach (i.e., the probability that a detected operation is correct) and the recall denotes the completeness of the approach (i.e., the probability that an actually applied operation is detected). Both values may range from 0 to 1, whereas a higher value is better than a lower one.

4.5 Results and discussion

In this section, we present the results of our experiment. We specifically discuss the results of our GA algorithm in terms of precision and recall and in terms of its stability. We also assess the effectiveness of our approach by comparing it to two other approaches. Finally, we discuss some threats to the validity of the results of our experiment.

4.5.1 Precision and recall

The precision and recall results might vary depending on the refactorings used, which are randomly generated, though guided by a meta-heuristic. We chose two projects (Xerces 2.5 and JHotDraw 5.2) to illustrate the results given by our approach. Figure 4.11 and Figure 4.12 show the results of multiple executions (31 executions) of our approach on Xerces and JHotDraw, respectively. Each of these figures displays the precision and the recall values for each execution.

Generally, the average precision and recall for all projects (84.6%) allows us to positively answer our first research question RQ1 and conclude that the results obtained by our approach are very encouraging. The precision, which is sometimes close to 100%, proves that all the refactorings proposed by our approach were indeed applied to the system's model in its subsequent version (i.e., the proposed refactorings match those returned by Ref-Finder when applied on the system's model and its subsequent version).

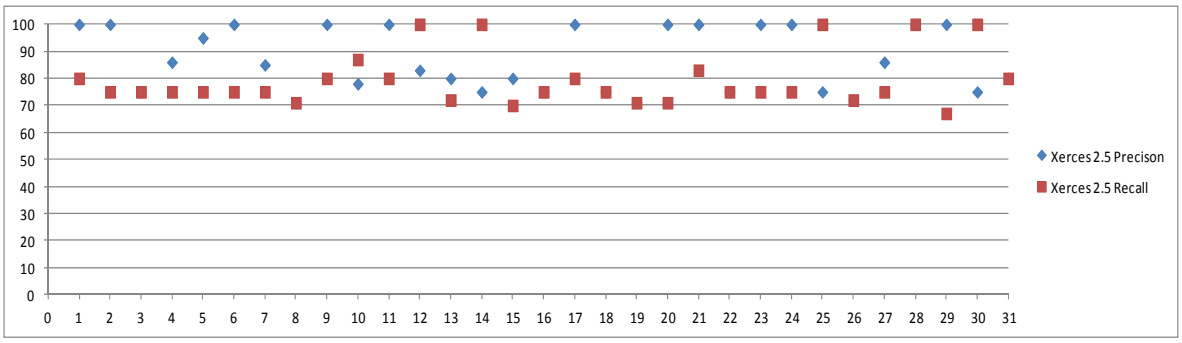


Figure 4.11 Multiple execution results for Xerces project

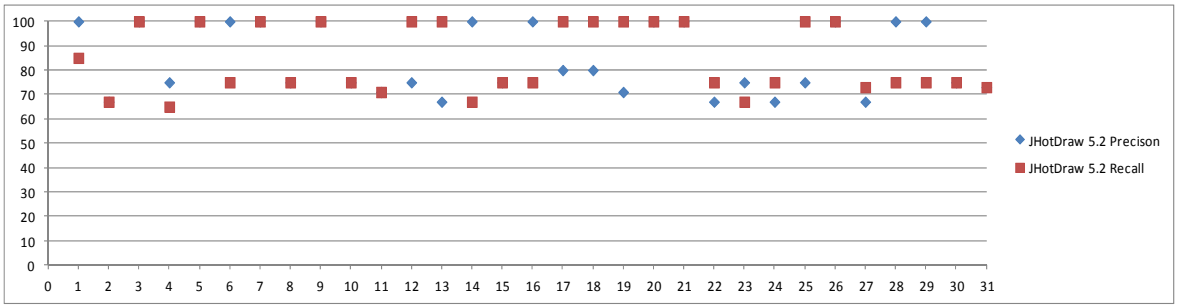


Figure 4.12 Multiple execution results for JHotDraw project

Despite the good results, we noticed a very slight decrease in recall versus precision in some projects; this is illustrated by Figure 4.11 for the Xerces project. We made a further analysis of the results to find out the factors behind this decline. Our analysis pointed out towards two important factors. The first factor is the project domain. In this study we tried to propose refactorings using a base of examples which contains different projects from different domains. We noticed that some projects focus on some types of refactorings compared to other (i.e., some projects in the base of examples have a big frequency of «*pull up field*» and «*pull up method*»). The second factor is the number and types of refactorings (i.e. twelve) considered in this experimentation. Indeed, we noticed that the refactorings («*pull up method*», «*pull up field*», «*add parameter*», «*extract class*», «*push down field*», «*push down method*», «*rename parameter*», «*rename method*» and «*move field*») are located correctly in our approach. We have no certainty that these factors can improve the results but we consider analyzing them as a future work to further clarify many issues.

4.5.2 Stability

To ensure that our results are relatively stable, we compared the results of multiple executions of the approach on each of the 8 open source projects. Figure 4.13 shows the precision results of these multiple executions for all the projects (see ANNEX II, p. 211) while Figure 4.14 shows the precision confidence intervals calculated on the precision average of these executions for each project. Similarly, Figure 4.15 shows the recall results of the 31 executions for all the projects while Figure 4.16 shows the recall confidence intervals calculated on the recall average of these executions for each project. The confidence intervals displayed by Figure 4.14 and Figure 4.16 confirm that precision and recall scores are approximately the same for different executions in all the projects in the base of examples. For example, the precision averages lies between these two bounds 73.76% and 80.56 for JGraph project and 92.77 and 98.46 for Ant project since 95% of confidence. This analysis allows us to conclude that our approach is stable which positively answers our second research question RQ2.

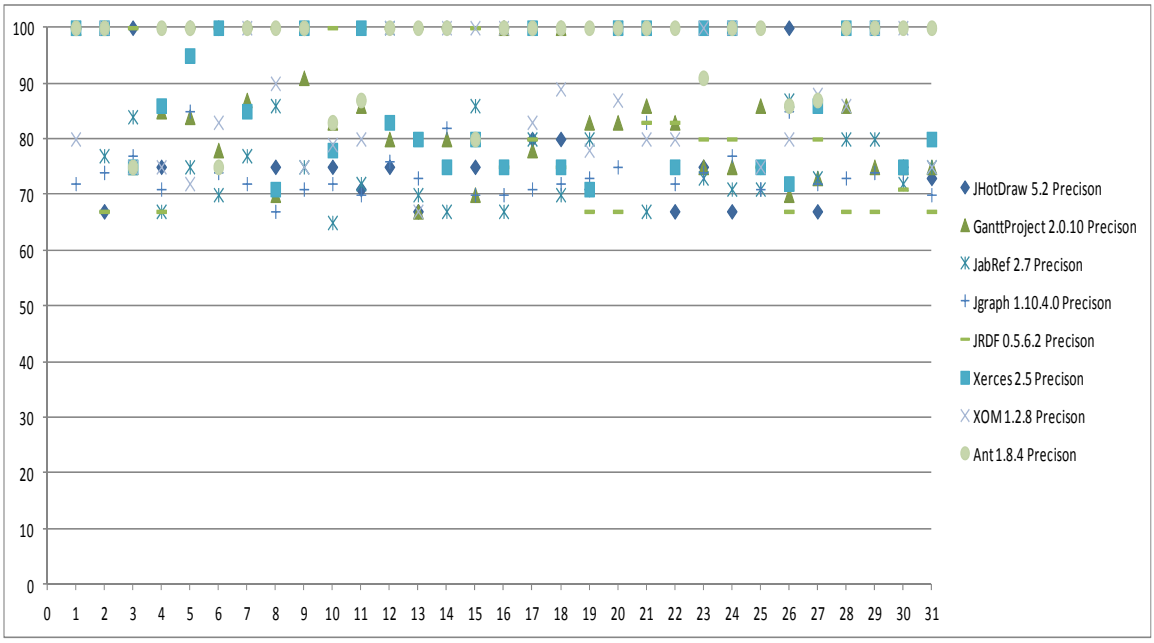


Figure 4.13 Multiple execution precison results of 8 open source projects in our approach

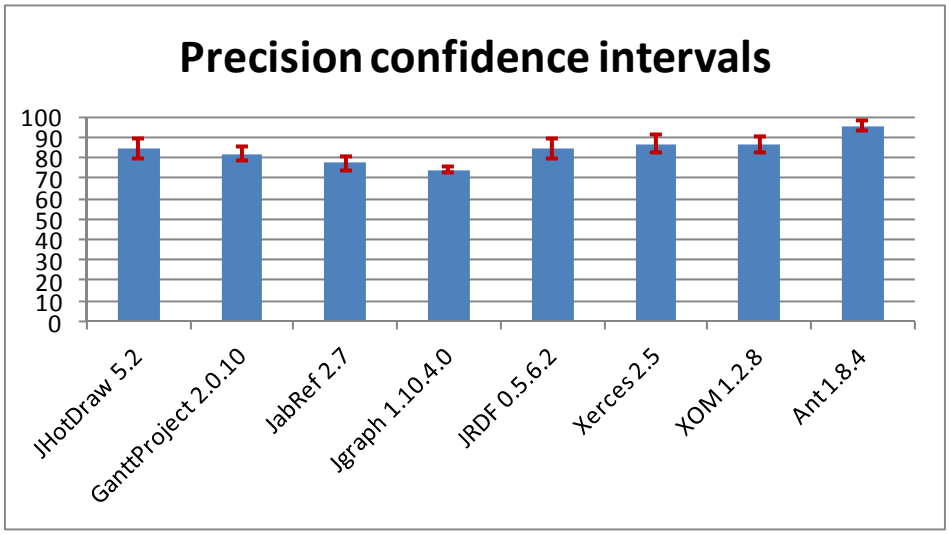


Figure 4.14 Confidence intervals for the precision average

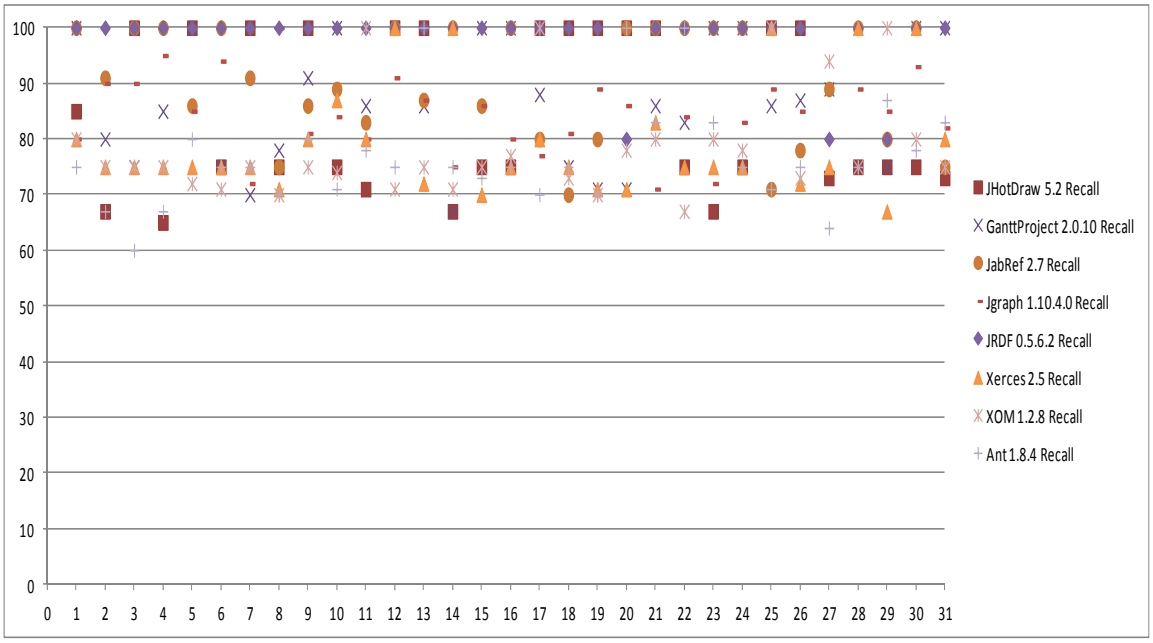


Figure 4.15 Multiple execution recall results of 8 open source projects in our approach

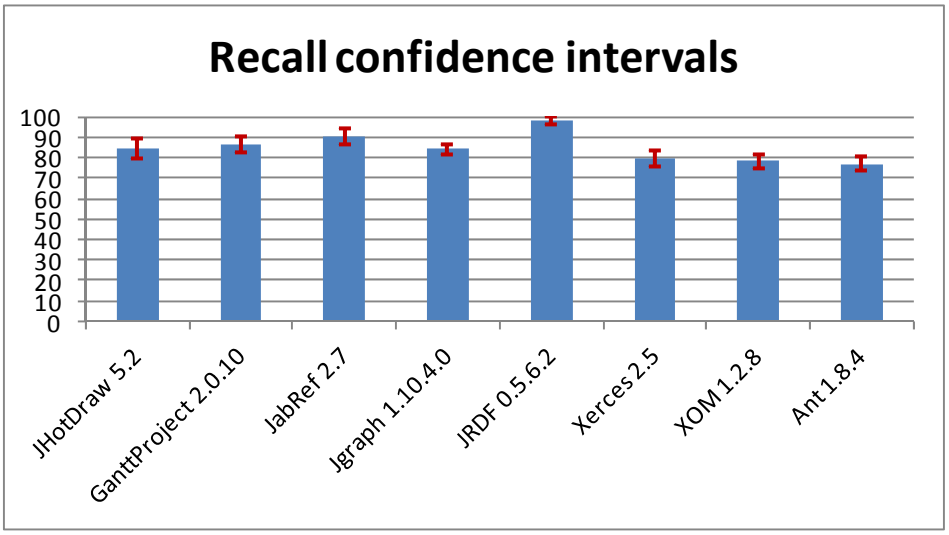


Figure 4.16 Confidence intervals for the recall average

4.5.3 Effectiveness of our approach

To assess the effectiveness of our approach, we conducted a comparative study between our approach and two other approaches: 1) a random search approach and 2) the algorithm proposed by Kessentini et al. in (Kessentini et al., 2011a). For the purpose of the first comparison, we implemented an algorithm that randomly selects pairs of CIM/CBEs. We run the random search algorithm under the same conditions in which we performed the experiment with our approach. Figure 4.17 and Figure 4.18 illustrate the results of multiple executions (31 executions) of the random search algorithm on the same 8 projects we used in our experiment. While the average precision and recall of our approach is around 85%, both precision and recall values of the 31 executions of the random search algorithm do not exceed 50%; i.e. these values vary between 20% and 50%. We consequently conclude that our approach is more effective than an equivalent random search approach.

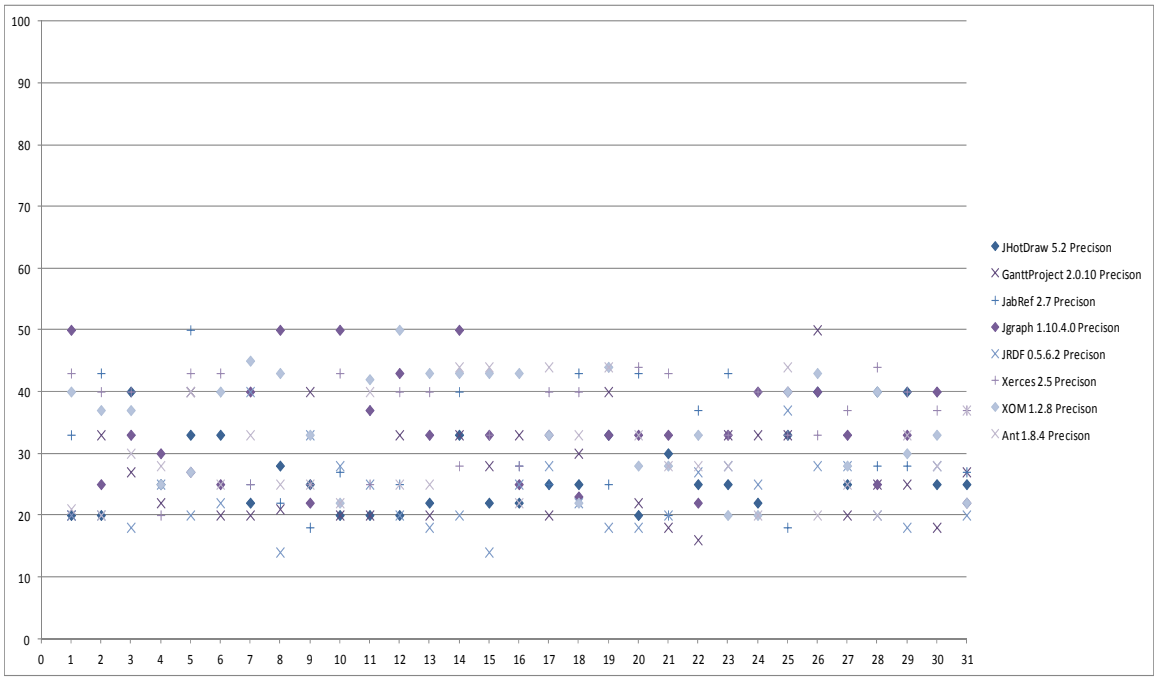


Figure 4.17 Precision of multiple executions of the random search algorithm

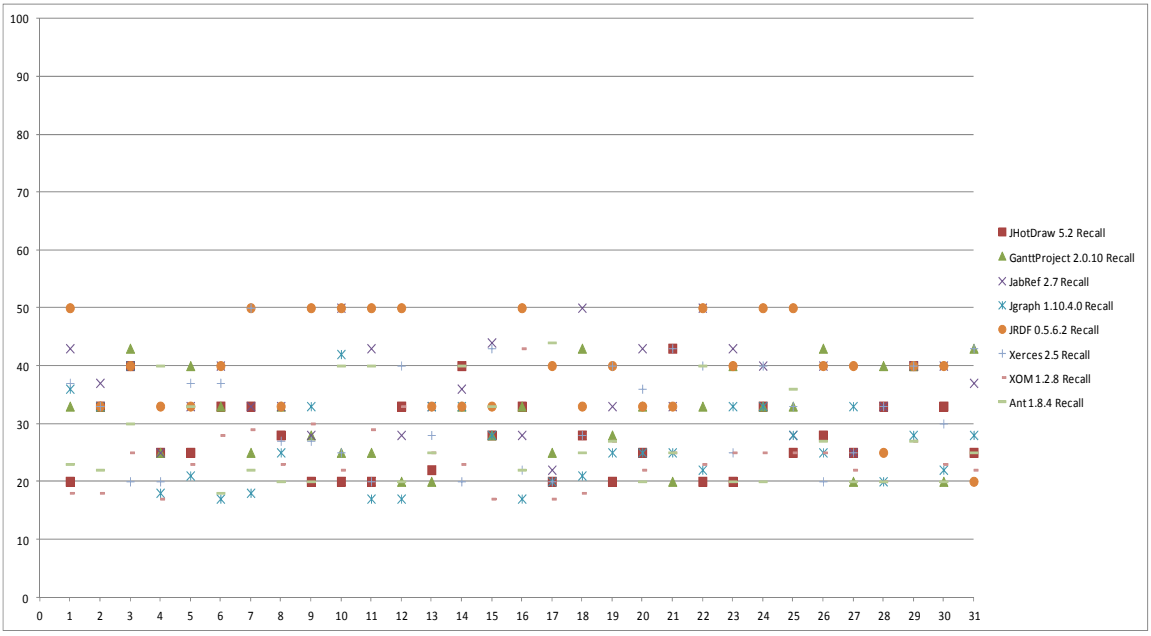


Figure 4.18 Recall of multiple executions of the random search algorithm

We also compared our approach to the approach proposed in (Kessentini et al., 2011a) where genetic programming (GP) is used to generate detection rules based on quality metrics. In fact, since the used algorithms are meta-heuristics thus they can produce different results on every run when applied to the same problem instance. To this end, the use of rigorous statistical tests is essential to provide support to the conclusions derived by analyzing such data (Arcuri and Briand, 2012). For this reason, we independently performed 31 executions using the algorithm in (Kessentini et al., 2011a) for each of the 8 open source projects that we used in our experiment, and we used the p-values of the Wilcoxon rank sum test (Wilcoxon, 1945) as a statistical test to compare the results of the two algorithms. . In our context, a p-value that is less than or equal to α (=0.05) means that the distributions of the results of the two algorithms are different in a statistically significant way. In fact, we computed the p-value of GP obtained results compared with our approach. In this way, we could decide whether the outperformance of our approach over the GP approach is statistically significant. Table 4.6 displays the precision and recall median values of our algorithm (MOREX) and the GP algorithm for the 8 open source projects. The p-value for the precision median results of GP compared with our approach is 0.0188 while the p-value

of the recall median results of GP compared with our approach is 0.0181. Consequently, as these values are less than α (= 0.05), we conclude that the precision and recall median values of our algorithm are statistically different from the GP ones on each of the systems. As Table 4.6 shows, it is clear that MOREX outperforms the approach in (Kessentini et al., 2011a) over all the open source systems.

Table 4.6 Precision and recall median values of GP (Kessentini et al., 2011a) and MOREX over 31 independent simulation runs

Models	*Precision MOREX (%)*	*Precision GP (%)*	*Recall MOREX (%)*	*Recall GP (%)*
Ant 1.8.4	78	72	81	77
GanttProject 2.0.10	82	78	84	82
JabRef 2.7	84	82	79	71
JGraphx 1.10.4.0	87	82	84	82
JHotDraw 5.2	86	81	86	81
JRDF v0.5.6.2	81	79	81	77
Xerces 2.5	82	77	83	81
Xom 1.2.8	86	79	87	78

4.5.4 Threats to validity

We have some points that we consider as threats to the generalization of our approach. The most important one is the use of the Ref_finder Tool to build the base of examples and at the same time we compare the results obtained by our algorithm to those given by Ref_finder. Another factor that could have been of influence on the obtained results is the sets of metrics and refactorings that we considered in our experiment. We made a preliminary analysis to select refactorings that apply at the model-level and we accordingly choose a set of related metrics. However further analysis is needed to build a catalog of refactorings that apply at to design models and to identify metrics that are impacted by these refactorings.

An important consideration is the impact of the example base size on the quality of refactoring solutions. In general, our approach does not need a large number of examples to obtain good detection results. The reliability of the proposed approach requires an example

set of applied refactoring on different systems. It can be argued that constituting such a set might require more work than these examples. In our study, we showed that by using some open source projects the approach can be used out of the box and will produce good refactoring results for the studied systems. However, we agree that, sometimes, within specific contexts it is difficult to define and find opportunities of refactorings. In an industrial setting, we could expect a company to start with some few open source projects, and gradually migrate its set of refactoring examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

Finally, since we viewed the model refactorings' generation problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed our algorithm on a standard desktop computer (i7 CPU running at 2.67 GHz with 8GB of RAM). The execution time for refactorings' generation with a number of iterations (stopping criteria) fixed to 1000 was less than three minutes. This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of refactorings and the size of the models in the base of examples.

4.6 Related work

Much work has been done on source code refactoring. The best way to correct the source code is to analyse it and to propose the appropriate refactorings to correct the defects it may contain (Fowler and Beck, 1999). This method is very expensive in terms of time and resources. Consequently many approaches were proposed to (semi)automatically support source code refactoring (e.g., (Du Bois et al., 2004), (Moha et al., 2010), (Liu et al., 2009) and (Kataoka et al., 2001)). These approaches use different techniques and strategies. For example, the work in (Du Bois et al., 2004) analyzed the best and worst-case impact of refactorings on coupling and cohesion dimensions. Most of the considered refactorings are applied at the code source level (e.g., Move Method, Replace Method with Method Object,

Replace Data Value with Object, and Extract Class). The approach in (Moha et al., 2010) proposed to represent code smells and use these representations to generate appropriate refactoring rules that can be automatically applied to source code. In (Kataoka et al., 2001), program invariants are used to detect a specific point in the program to apply refactoring, and an invariant pattern matcher was developed and used on an existing Java code base to suggest some common refactorings.

Model refactoring is still at a relatively young stage of development. Most of existing approaches for automating refactoring activities at the model-level are based on rules that can be expressed as assertions (i.e., invariants, pre-and post-condition) (Ragnhild et al., 2007; Van Kempen et al., 2005), or graph transformations targeting refactoring operations in general (e.g., (Biermann, 2010; Mens et al., 2007b)) or refactorings related to design patterns' applications (e.g., (El-Boussaidi and Mili, 2011)). The use of invariants (Ragnhild et al., 2007) has been proposed to detect some parts of the model that require refactoring. Refactorings are expressed using declarative rules. However, a complete specification of refactorings requires an important number of rules and the refactoring rules must be complete, consistent, non-redundant and correct. In (El-Boussaidi and Mili, 2011) refactoring rules are used to specify design patterns' applications. In this context, design problems solved by these patterns are represented using models and the refactoring rules transform these models according to the solutions proposed by the patterns. However, not all design problems are representable using models; i.e., for some patterns, the problem space is quite large and the problem cannot be captured in a single, or a handful of problem models (El-Boussaidi and Mili, 2011). Finally an issue that is common to most of these approaches is the problem of sequencing and composing refactoring rules. This is related to the control of rules' applications within rule-based transformational approaches in general.

Our approach is inspired by contributions in search-based software engineering (SBSE) (e.g. (O'Keeffe, 2008), (Harman and Tratt, 2007), (Kessentini et al., 2012), (Seng et al., 2006) and (Jensen and Cheng, 2010)). As the name indicates, SBSE uses a search-based approach to solve optimization problems in software engineering. Techniques based on SBSE are a good

alternative to tackle many of the above mentioned issues (Kessentini et al., 2012). For example, a heuristic-based approach is presented in (Harman and Tratt, 2007; O'Keeffe, 2008) in which various software measures are used as indicators for the need of a certain refactoring. In (Seng et al., 2006), a genetic algorithm is used to suggest refactorings to improve the class structure of a system. The algorithm uses a fitness function that relies on a set of existing object oriented metrics. Harman and Tratt (Harman and Tratt, 2007) propose to use the Pareto optimality concept to improve search-based refactoring approaches when the evaluation function is based on a weighted sum of metrics. Both the approaches in (Seng et al., 2006) and (Harman and Tratt, 2007) were limited to the Move Method refactoring operation. In (O'Keeffe, 2008), the authors present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics. The results of the experiments on five open-source systems showed that hill-climbing performs better than the other algorithms. In (Jensen and Cheng, 2010), the authors proposed an automated refactoring approach that uses genetic programming (GP) to support the composition of refactorings that introduce design patterns. The fitness function used to evaluate the applied refactorings relies on the same set of metrics as in (O'Keeffe, 2008) and a bonus value given for the presence of design patterns in the refactored design. Our approach can be seen as linked to this approach as we aim at proposing a combination of refactorings that must be applied to a design model. Our work is more related to the work in (Kessentini et al., 2012) where the authors proposed a by-example approach based on search-based techniques for model transformation. A Particle Swarm Optimization (PSO) algorithm is used to find the best subset of transformation fragments in the base of examples, that can be used to transform a source model (i.e., Class Diagram) to a target model (i.e., Relational Schema). Hence, this approach targets exogenous transformations (i.e., different source and target languages) while our proposal MOREX is dedicated to refactorings which are endogenous transformations that aim at correcting design defects. Furthermore, the fitness function proposed in (Kessentini et al., 2012) relies on the adequate mapping of the selected transformation examples with the constructs of the model (e.g., class, relationship) to be transformed while our fitness function exploits the structural similarity between classes. To conclude, in our contribution we propose to use a different

metaheuristic algorithm to a different problem than the one in (Kessentini et al., 2012) with a new adaptation (fitness function, change operators, etc.).

4.7 Conclusion and future work

In this paper we introduced MOREX (MOdel REfactoring by eXample), an approach to automate model refactoring using heuristic-based search. The approach considers the refactoring as an optimization problem and it uses a set of refactoring examples to propose appropriate sequences of refactorings that can be applied on a source model. MOREX randomly generates sequences of applicable refactorings and evaluates their quality depending on the similarity between the source model and the examples of models at hand.

We have evaluated our approach on real-world models extracted from eight open source systems. The experimental results indicate that the proposed refactorings are comparable to those expected, i.e., the proposed refactorings match those returned by the Ref-Finder tool when applied on a model and its subsequent version. We also performed multiple executions of the approach on the 8 open source projects and the results have shown that the approach is stable regarding its precision and recall.

While the results of the approach are very promising, we plan to extend it in different ways. One issue that we want to address as a future work is related to the base of examples. In the future we want to extend our base of examples to include more refactoring operations. We also want to study and analyze the impact of using domain-specific examples on the quality of the proposed sequences of refactorings. Actually, we kept the random aspect that characterizes genetic algorithms even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to correct the one under analysis.

We also plan to compare our results with other existing approaches other than the Ref-Finder tool and perform a further analysis on the nature and type of refactorings that are easier or

harder to detect. In addition, the evaluation of the sequences of refactorings returned by our approach was based on the similarity between the classes of the source model and the classes from the base of examples. However, only the syntactic aspect was considered when computing these similarities, i.e., the similarity was based on a set of metrics that are mostly related to the structural features of the classes (e.g., number of attributes, number of methods, etc.). In the future, we plan to study the semantic properties (e.g., similarity of classes' names) that can be used as similarity or dissimilarity factors to enhance our evaluation function.

We noticed that majority of search based refactoring approaches ((O'Keeffe, 2008), (Harman and Tratt, 2007), (Ben Fadhel et al., 2012) and also our MOREX approach (Ghannem et al., 2014c) presented in the current chapter) have defined the fitness function as a combination of software metrics. Indeed, the fact that the values of some metrics were improved after some refactorings does not necessarily mean or ensure that these refactorings make sense. This observation was at the origin of our next two chapters (chapter 5 and chapter 6). In order to give sense to the suggested refactorings generated by our MOREX approach, we had the idea to put the designer in the loop by adapting the interactive genetic algorithm. This contribution is more detailed in the next chapter.

CHAPTER 5

MODEL REFACTORING USING INTERACTIVE GENETIC ALGORITHM

Adnane Ghannem, Ghizlane El Boussaidi[1] and Marouane Kessentini[2]

[1]Department of Software and IT Engineering, École de Technologie Supérieure,

1100 Notre-Dame West, Montreal, Quebec, (H3C 1K3) Canada

[2]Department of Computer and Information Science, University of Michigan - Dearborn

4901 Evergreen Road, Dearborn, MI 48128 USA

This paper has been published in Symposium on Search-Based

Software Engineering

ABSTRACT

Refactoring aims at improving the quality of design while preserving its semantic. Providing an automatic support for refactoring is a challenging problem. This problem can be considered as an optimization problem where the goal is to find appropriate refactoring suggestions using a set of refactoring examples. However, some of the refactorings proposed using this approach do not necessarily make sense depending on the context and the semantic of the system under analysis. This paper proposes an approach that tackles this problem by adapting the Interactive Genetic Algorithm (IGA) which enables to interact with users and integrate their feedbacks into a classic GA. The proposed algorithm uses a fitness function that combines the structural similarity between the analyzed design model and models from a base of examples, and the designers' ratings of the refactorings proposed during execution of the classic GA. Experimentation with the approach yielded interesting and promising results.

Keywords: Software maintenance, Interactive Genetic Algorithm, Model refactoring, Refactoring by example.

5.1 Introduction

Software maintenance is considered the most expensive activity in the software system lifecycle (Lientz et al., 1978). Maintenance tasks can be seen as incremental modifications to a software system that aim to add or adjust some functionality or to correct some design flaws. However, as the time goes by, the system's conceptual integrity erodes and its quality degrades; this deterioration is known in the literature as the software decay problem (Fowler, 1999). A common and widely used technique to cope with this problem is to continuously restructure the software system to improve its structure and design. The process of restructuring object oriented systems is commonly called refactoring (Mens and Tourwé, 2004). According to Fowler (Fowler, 1999), refactoring is the disciplined process of cleaning up code to improve the software structure while preserving its external behavior. Many researchers have been working on providing support for refactoring operations (e.g., (Opdyke, 1992), (Fowler, 1999), and (Moha, 2008)). Existing tools provide different environments to manually or automatically apply refactoring operations to correct, for example, code smells. Indeed, existing work has, for the most part, focused on refactorings at the source code level. Actually, the rise of the model-driven engineering (MDE) approach increased the interest and the needs for tools supporting refactoring at the model-level. In MDE, abstract models are successively refined into more concrete models, and a model refactoring tool will be of great value within this context.

The search-based refactoring approaches proved their effectiveness to propose refactorings to improve the model's design quality. They adapted some of the known heuristics methods (e.g. Simulated annealing, Hill_climbing) as proposed in (Harman and Tratt, 2007; O'Keeffe, 2008; O'Keeffe and O'Cinneide, 2006) and Genetic Algorithms as in (Kessentini et al., 2008). These approaches relied, for the most part, on a combination of quality metrics to formulate their optimization goal (i.e., the fitness function). A major problem founded in these approaches is that the quality metrics consider only the structural properties of the system under study; the semantic properties of the system are not considered. In this context, Mens and Tourwé (Mens and Tourwé, 2004) argue that most of the refactoring tools cannot offer a

full-automatic support because part of the necessary knowledge−especially those related to the semantics−for performing the refactoring remains implicit in designers' heads. Indeed, recognizing opportunities of model refactoring remains a challenging issue that is related to the model marking process within the context of MDE which is a notoriously difficult problem that requires design knowledge and expertise (El-Boussaidi and Mili, 2008).

To take into account the semantics of the software system, we propose a model refactoring approach based on an Interactive Genetic Algorithm (IGAs) (Takagi, 2001). Two types of knowledge are considered in this approach. The first one comes from the examples of refactorings. For this purpose, we hypothesize that the knowledge required to propose appropriate refactorings for a given object-oriented model may be inferred from other existing models' refactorings when there is some structural similarities between these models and the given model. From this perspective, the refactoring is seen as an optimization problem that is solved using a Genetic Algorithm (GA). The second type of knowledge comes from the designer's knowledge. For this purpose, the designer is involved in the optimization process by continuously interacting with the GA algorithm; this enables to adjust the results of the GA progressively exploiting the designer's feedback. Hence the proposed approach (MOREX+I: MOdel REfactoring by eXample plus Interaction) relies on a set of refactoring examples and designer's feedbacks to propose sequences of refactorings. MOREX+I takes as input an initial model, a base of examples of refactored models and a list of metrics calculated on both the initial model and the models in the base of examples, and it generates as output a solution to the refactoring problem. In this paper, we focus on UML class diagrams. In this case, a solution is defined as a sequence of refactorings that maximize as much as possible the similarity between the initial and revised class diagrams (i.e., the class diagrams in the base of examples) while considering designer's feedbacks.

The primary contributions of the paper are 3-fold: 1) We introduce a model refactoring approach based on the use of examples. The approach combines implicitly the detection and the correction of design defects at the model-level by proposing a sequence of refactorings that must be applied on a given model. 2) We use the IGA to allow the integration of

feedbacks provided by designers upon solutions produced during the GA evolution. 3) We report the results of an evaluation of our approach.

The paper is organized as follows. Section 5.2 is dedicated to the background where we introduce some basic concepts and the related work. The overall approach is described in section 5.3. Section 5.4 reports on the experimental settings and results, while section 5.5 concludes the paper and outlines some future directions to our work.

5.2 Background

5.2.1 Class diagrams refactorings and quality metrics

Model refactoring is a controlled technique for improving the design (e.g., class diagrams) of an existing model. It involves applying a series of small refactoring operations to improve the design quality of the model while preserving its behavior. Many refactorings were proposed and codified in the literature (see e.g., (Fowler, 1999)). In our approach, we consider a subset of the 72 refactorings defined in (Fowler, 1999); i.e., only those refactorings that can be applied to UML class diagrams. Indeed, some of the refactorings in (Fowler, 1999) may be applied on design models (e.g. *Move_Method, Rename_method, Move_Attribute, Extract_Class* etc.) while others cannot be (e.g. *Extract_Method, Inline_Method, Replace_Temp_With_Query* etc.). In our approach we considered a list of twelve refactorings (e.g. *Extract_class, Push_down_method, Pull_up_method,* etc.) based on (Fowler, 1999). The choice of these refactorings was mainly based on two factors: 1) they apply at the class diagram-level; and 2) they can be link to a set of model metrics (i.e., metrics which are impacted when applying these refactorings).

Metrics provide useful information that help assessing the level of conformance of a software system to a desired quality (Fenton and Pfleeger, 1998). Metrics can also help detecting some similarities between software systems. The most widely used metrics for class diagrams are the ones defined by Genero et al. (Genero et al., 2002). In the context of our approach, we

used a list of sixteen metrics (e.g. *Number of attributes: NA, Number of methods: NMeth, Number of dependencies: NDep*, etc.) including the eleven metrics defined in (Genero et al., 2002) to which we have added a set of simple metrics (e.g., *number of private methods in a class, number of public methods in a class*). All these metrics are related to the class entity which is the main entity in a class diagram.

5.2.2 Interactive Genetic Algorithm (IGA)

Heuristic search are serving to promote discovery or learning (Pearl, 1984). There is a variety of methods which support the heuristic search as hill_climbing (Mitchell, 1998), genetic algorithms (GA) (Goldberg, 1989), etc. GA is a powerful heuristic search optimization method inspired by the Darwinian theory of evolution (Koza, 1992). The basic idea behind GA is to explore the search space by making a population of candidate solutions, also called individuals, evolve toward a "good" solution of a specific problem. Each individual (i.e., a solution) of the population is evaluated by a fitness function that determines a quantitative measure of its ability to solve the target problem. Exploration of the search space is achieved by selecting individuals (in the current population) that have the best fitness values and evolving them by using genetic operators, such as crossover and mutation. The crossover operator insures generation of new children, or offspring, based on parent individuals while the mutation operator is applied to modify some randomly selected nodes in a single individual. The mutation operator introduces diversity into the population and allows escaping local optima found during the search. Once selection, mutation and crossover have been applied according to given probabilities, individuals of the newly created generation are evaluated using the fitness function. This process is repeated iteratively, until a stopping criterion is met. This criterion usually corresponds to a fixed number of generations.

Interactive GA (IGAs) (Dawkins, 1986) combines a genetic algorithm with the interaction with the user so that he can assign a fitness to each individual. This way IGA integrates the user's knowledge during the regular evolution process of GA. For this reason, IGA can be used to solve problems that cannot be easily solved by GA (Kim and Cho, 2000). A variety

of application domains of IGA include development of fashion design systems (Kim and Cho, 2000), music composition systems (Chen, 2007), software re-modularization (Bavota et al., 2012) and some other IGAs' applications in other fields (Takagi, 2001). One of the key elements in IGAs is the management of the number of interactions with the user and the way an individual is evaluated by the user.

5.2.3 Related work

Model refactoring is still at a relatively young stage of development compared to the work that has been done on source-code refactoring. Most of existing approaches for automating refactoring activities at the model-level are based on rules that can be expressed as assertions (i.e., invariants, pre-and post-conditions) (Ragnhild et al., 2007; Van Kempen et al., 2005), or graph transformations targeting refactoring operations in general (Biermann, 2010; Mens et al., 2007b) or design patterns' applications in particular (e.g., (El-Boussaidi and Mili, 2011)). In (Ragnhild et al., 2007) invariants are used to detect some parts of the model that require refactoring and the refactorings are expressed using declarative rules. However, a complete specification of refactorings requires an important number of rules and the refactoring rules must be complete, consistent, non-redundant and correct. In (El-Boussaidi and Mili, 2011) refactoring rules are used to specify design patterns' applications. In this context, design problems solved by these patterns are represented using models and the refactoring rules transform these models according to the solutions proposed by the patterns. However, not all design problems are representable using models. Finally an issue that is common to most of these approaches is the problem of sequencing and composing refactoring rules. This is related to the control of rules' applications within rule-based transformational approaches in general.

Our approach is inspired by contributions in search-based software engineering (SBSE) (e.g. (Harman and Tratt, 2007; Jensen and Cheng, 2010; Kessentini et al., 2008; O'Keeffe, 2008; Seng et al., 2006)). Techniques based on SBSE are a good alternative to tackle many of the above mentioned issues (Kessentini et al., 2008). For example, a heuristic-based approach is

presented in (Harman and Tratt, 2007; O'Keeffe, 2008; Seng et al., 2006) in which various software metrics are used as indicators for the need of a certain refactoring. In (Seng et al., 2006), a genetic algorithm is used to suggest refactorings to improve the class structure of a system. The algorithm uses a fitness function that relies on a set of existing object oriented metrics. Harman and Tratt (Harman and Tratt, 2007) propose to use the Pareto optimality concept to improve search-based refactoring approaches when the evaluation function is based on a weighted sum of metrics. Both the approaches in (Seng et al., 2006) and (Harman and Tratt, 2007) were limited to the Move Method refactoring operation. In (O'Keeffe, 2008), the authors present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics. The results of the experiments on five open-source systems showed that hill-climbing performs better than the other algorithms. In (Jensen and Cheng, 2010), the authors proposed an automated refactoring approach that uses genetic programming (GP) to support the composition of refactorings that introduce design patterns. The fitness function used to evaluate the applied refactorings relies on the same set of metrics as in [12] and a bonus value given for the presence of design patterns in the refactored design. Our approach can be seen as linked to this approach as we aim at proposing a combination of refactorings that must be applied to a design model. Our approach was inspired by the work in (Bavota et al., 2012) where the authors apply an Interactive Genetic Algorithm to the re-modularization problem which can be seen as a specific subtype of the refactoring problem. Our work is also related to the approach in (Kessentini et al., 2012) where the authors apply an SBSE approach to model transformations. However this approach focuses on general model transformations while our focus is on refactorings which are commonly codified transformations that aim at correcting design defects.

To conclude, most of the approaches that tackled the refactoring as an optimization problem by the use of some heuristics suppose, to some extent, that a refactoring operation is appropriate when it optimizes the fitness function (FF). Most of these approaches defined their FF as a combination of quality metrics to approximate the quality of a model. However, refactoring operations are design transformations which are context-sensitive. To be

appropriately used, they require some knowledge of the system to be refactored. Indeed, the fact that the values of some metrics were improved after some refactorings does not necessarily mean or ensure that these refactorings make sense. This observation is at the origin of the work described in this paper as described in the next section.

5.3 Heuristic Search Using Interactive Genetic Algorithm

5.3.1 Interactive Genetic Algorithm adaptation

The approach proposed in this paper exploits examples of model refactorings, a heuristic search technique and the designer's feedback to automatically suggest sequences of refactorings that can be applied on a given model (i.e., a UML class diagram). A high-level view of our adaptation of IGA to the model refactoring problem is given in Algorithm 5.1. The algorithm takes as input a set of quality metrics, a set of model refactoring examples, a percentage value corresponding to the percentage of a population of solutions that the designer is willing to evaluate, the maximum number of iterations for the algorithm and the number of interactions with the designer. First, the algorithm runs classic GA (line 2) for a number of iterations (i.e., the maximum number of iterations divided by the number of interactions). Then a percentage of solutions from the current population is selected (line 3). In lines 4 to 7, we get designers' feedbacks for each refactoring in each selected solution and we update their fitness function. We generate a new population (p+1) of individuals (line 8) by iteratively selecting pairs of parent individuals from population p and applying the crossover operator to them; each pair of parent individuals produces two children (solutions). We include both the parent and child variants in the new population. Then we apply the mutation operator, with a probability score, for both parent and child to ensure the solution diversity; this produces the population for the next generation. The algorithm terminates when the maximum iteration number is reached, and returns the best set of refactorings' sequences (i.e., best solutions from all iterations).

```
Input: Set of quality metrics
Input: Set of model refactoring examples
Input: Percentage (P%)
Input: MaxNbrIterations
Input: NbrOfInteractions
Output: A sequence of refactorings

1   for i = 1 . . . NbrOfInteractions do
2     Evolve GA for NbrIterations
3     Select P% of best solutions from the current population.
4     for-each selected solution do
5         Ask the designer whether each refactoring within the selected solution makes sense.
6         Update the FF of the selected solution to integrate the feedback.
7     end for-each
8     Create a new GA population using the updated solutions
9   end for
10  Continue (non-interactive) GA evolution until it converges or it reaches maxNbrIterations
```

Algorithm 5.1 High-level pseudo-code for IGA adaptation to our problem

In the following subsections we present the details of the regular GA adaptation to the problem of generating refactoring sequences and how we collect the designers' feedbacks and integrate it in the fitness function computation

5.3.2 Representing an individual and generating the Initial Population

An individual (i.e., a candidate solution) is a set of blocks. The upper part of Figure 5.1 shows an individual with three blocks. The first part of the block contains the class (e.g. Order) chosen from the initial model (model under analysis) called CIM, the second part contains the class (e.g Person) from the base of examples that was matched to CIM called CBE, and finally the third part contains a list of refactorings (e.g. *Pull_Up_Method(calc_taxes(), LineOrder, Orde)*) which is a subset of the refactorings that were applied to CBE (in its subsequent versions) and that can be applied to CIM. In our approach, classes from the model (CIMs) and the base of examples (CBEs) are represented

using predicates that describe their attributes, methods and relationships. In addition, the representation of a CBE class includes a list of refactorings that were applied to this class in a subsequent version of the system's model to which CBE belongs. The subset of a CBE subsequent refactorings that are applicable to a CIM class constitutes the third part of the block having CIM as its first part and CBE as its second part. Hence, the selection of the refactorings to be considered in a block is subjected to some constraints to avoid conflicts and incoherence errors. For example, if we have a *Move_attribute* refactoring operation in the CBE class and the CIM class doesn't contain any attribute, then this refactoring operation is discarded as we cannot apply it to the CIM class.

Hence the individual represents a sequence of refactoring operations to apply and the classes of the initial model on which they apply. The bottom part of Figure 5.1 shows the fragments of an initial model before and after the refactorings proposed by the individual (at the top of the figure) were applied.

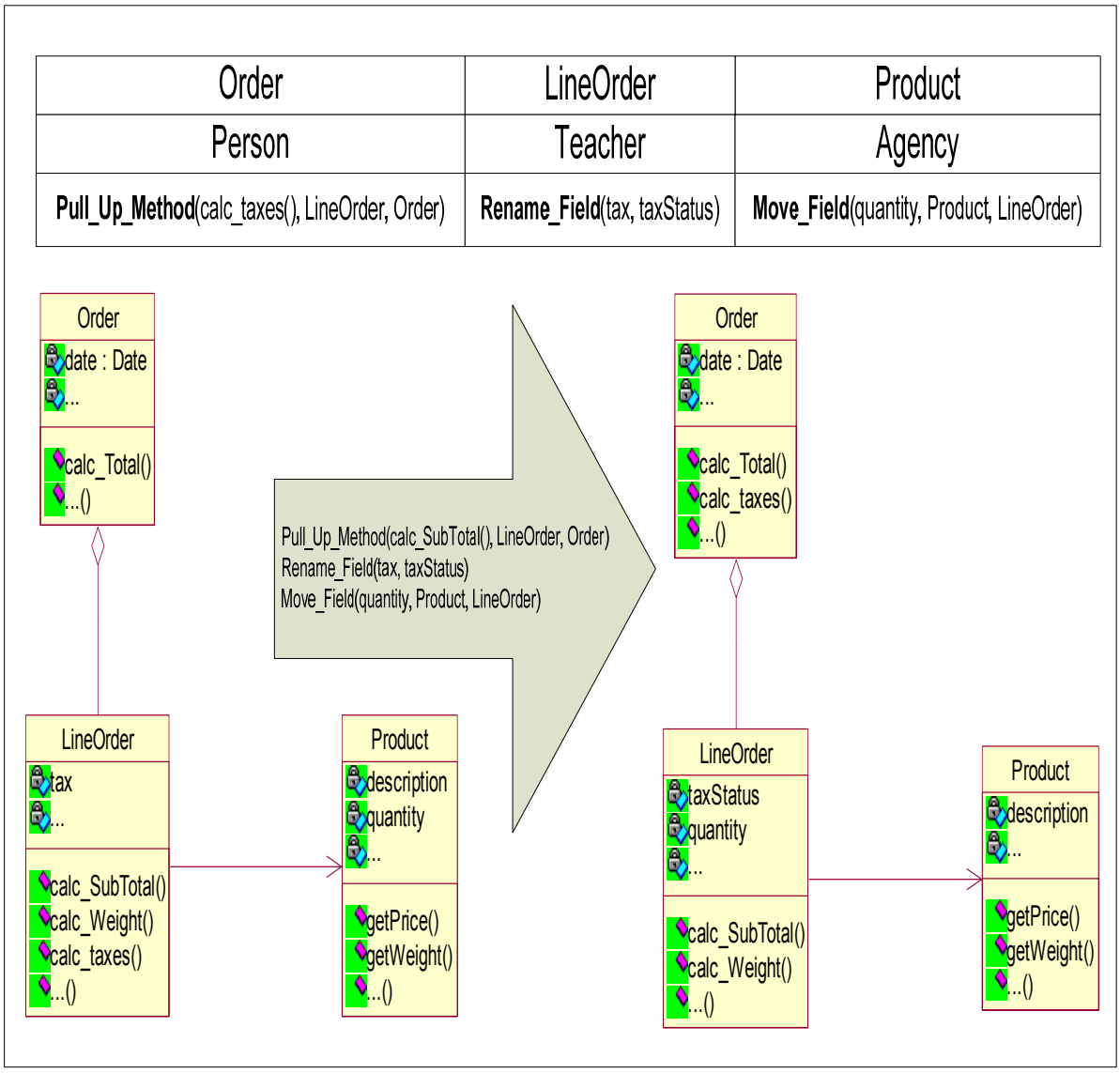


Figure 5.1 Individual representation

To generate an initial population, we start by defining the maximum individual size. This parameter can be specified either by the user or randomly. Thus, the individuals have different sizes. Then, for each individual we randomly assign: 1) a set of classes from the initial model that is under analysis and their matched classes from the base of examples, and 2) a set of refactorings that we can possibly apply on the initial model class among the refactorings proposed from the base of examples class.

Genetic operators

Selection

To select the individuals that will undergo the crossover and mutation operators, we used the stochastic universal sampling (SUS) (Koza, 1992), in which the probability of selection of an individual is directly proportional to its relative fitness in the population. For each iteration, we use SUS to select 50% of individuals from population p for the new population p+1. These (population_size/2) selected individuals will "give birth" to another (population_size/2) new individuals using crossover operator.

Crossover

For each crossover, two individuals are selected by applying the *SUS* selection (Koza, 1992). Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring *p'1* and *p'2* from the two selected parents *p1* and *p2* as follows: A random position, *k*, is selected. The first *k* refactorings of *p1* become the first *k* elements of *p'2*. Similarly, the first *k* refactorings of *p2* become the first *k* refactorings of *p'1*. The rest of refactorings (from position k+1 until the end of the sequence) in each parent *p1* and *p2* are kept. For instance, Figure 5.2 illustrates the crossover operator applied to two individuals (parents) *p1* and *p2* where the position *k* takes the value *2*.

Mutation

The mutation operator consists of randomly changing one or more elements in the solution. Hence, given a selected individual, the mutation operator first randomly selects some refactorings among the refactoring sequence proposed by the individual. Then the selected refactorings are replaced by other refactorings. Figure 5.3illustrates the effect of a mutation on an individual.

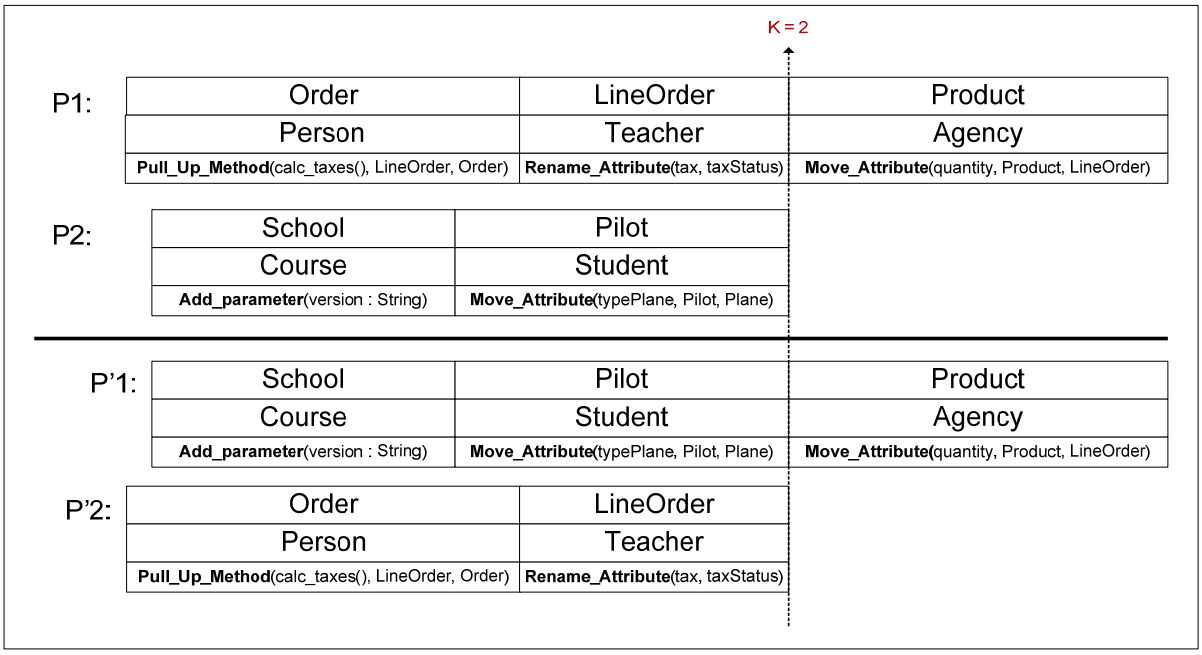


Figure 5.2 Crossover operator

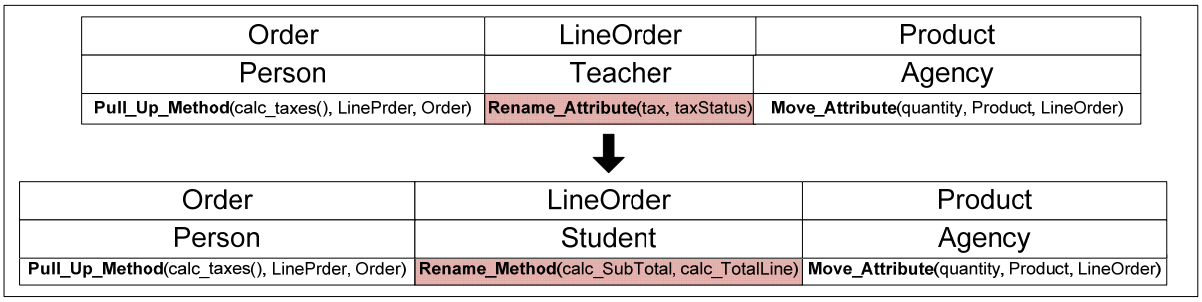


Figure 5.3 Mutation operator

5.3.3 Evaluating an individual within the Classic GA

The quality of an individual is proportional to the quality of the refactoring operations composing it. In fact, the execution of these refactorings modifies various model fragments; the quality of a solution is determined with respect to the expected refactored model. However, our goal is to find a way to infer correct refactorings using the knowledge that has been accumulated through refactorings of other models of past projects and feedbacks given by designers. Specifically, we want to exploit the similarities between the actual model and

other models to infer the sequence of refactorings that we must apply. Our intuition is that a candidate solution that displays a high similarity between the classes of the model and those chosen from the examples base should give the best sequence of refactorings. Hence, the fitness function aims to maximize the similarity between the classes of the model in comparison to the revised ones in the base of examples. In this context, we introduce first a similarity measure between two classes denoted by *Similarity* and defined by formula 6.1 and 6.2.

$$Similarity\ (CMI, CBE) = \frac{1}{m}\sum_{i=1}^{m} Sim(CMI_i, CBE_i) \tag{5.1}$$

$$Sim\ (CMI_i, CBE_i) = \begin{cases} 1 & if\ CMI_i = CBE_i \\ 0 & if\ CMI_i = 0\ or\ CBE_i = 0 \\ \dfrac{CMI_i}{CBE_i} & if\ CMI_i < CBE_i \\ \dfrac{CBE_i}{CMI_i} & if\ CBE_i < CMI_i \end{cases} \tag{5.2}$$

where ***m*** is the number of metrics considered in this project. CIM_i is the i^{th} metric value of the class CIM in the initial model while CBE_i is the i^{th} metric value of the class CBE in the base of examples. Using the similarity between classes, we define the fitness function of a solution, normalized in the range [0, 1], as:

$$f = \frac{1}{n}\sum_{j=1}^{n} Similarity(CMI_{Bj}, CBE_{Bj}) \tag{5.3}$$

where ***n*** is the number of blocks in the solution and CMI_{Bj} and CBE_{Bj} are the classes composing the first two parts of the j^{th} block of the solution. To illustrate how the fitness function is computed, we consider a system containing two classes as shown in Table 5.1 and

a base of examples containing two classes shown in Table 5.2. In this example we use six metrics and these metrics are given for each class in the model in Table 5.1 and each class of the base of examples in Table 5.2.

Table 5.1 Classes from the initial model and their metrics values

CMI	*NPvA*	*NPbA*	*NPbMeth*	*NPvMeth*	*NAss*	*NGen*
LineOrder	4	1	3	1	1	1
Product	2	2	6	0	1	0

Table 5.2 Classes from the base of examples and their metrics values

CBE	*NPvA*	*NPbA*	*NPbMeth*	*NPvMeth*	*NAss*	*NGen*
Student	2	1	3	0	3	0
Plane	5	1	4	0	1	0

Consider an individual/solution I_1 composed by two blocks (*LineOrder*/*Student* and *Product*/*Plane*). The fitness function of I_1 is calculated as follows:

$$\boldsymbol{f_{I_1}} = \frac{1}{12}\left[\left(\frac{2}{4}+1+1+0+\frac{1}{3}+0\right)+\left(\frac{2}{5}+\frac{1}{2}+\frac{4}{6}+0+1+0\right)\right] = 0{,}45 \qquad (5.4)$$

5.3.4 Collecting and Integrating the Feedbacks from Designers

Model refactoring is a design operation that is context-sensitive. In addition, depending on the semantics of the system under analysis and the system's evolution as foreseen by different designers, a refactoring proposed by the classic GA can be considered as mandatory by a designer and as acceptable by another. Even if a sequence of refactorings optimizes the fitness function (as defined in the previous section), that does not ensure that these refactorings conform to and preserve the semantics of the system. Consequently, we use Interactive GA (IGA) to partly tackle this problem by interacting with designers and getting their feedbacks on a number of the proposed refactoring sequences. To do so, we adopted a five level scale defined by (Saaty, 1985) to rate the proposed refactorings; i.e., we distinguish

five types of rating that a designer can assign to a proposed refactoring. The meaning and the value of each type of rating are as follows:

1. Critical (value = 1): it is mandatory to apply the proposed refactoring;
2. Desirable (value = 0.8): it is useful to apply the refactoring to enhance some aspect of the model but it's not mandatory;
3. Neutral (value = 0.5): the refactoring is applicable but the designer does not see it as necessary or desirable;
4. Undesirable (value = 0.3): the refactoring is applicable but it is not useful and could alter the semantics of the system;
5. Inappropriate (value = 0): the refactoring should not be applied because it breaks the semantics of the system.

As described in section 5.3.1., during the execution of IGA, the designer is asked to rate a percentage of the best solutions found by the classic GA after a defined number of iterations. For each of the selected solutions, the designer assigns a rating for each refactoring included in the solution. Depending on the values entered by the designer, we re-evaluate the global fitness function of the solution as follows. For each block of the solution, we compute the block rating as an average of the ratings of the refactorings in the block. Then we compute the overall designer's rating as an average of all blocks ratings. Finally, the new fitness function of the solution is computed as an average of its old fitness function and the overall designer's rating. The new values of the fitness functions of the selected solutions are injected back into the IGA process to form a new population of individuals.

5.4 Experiments

The goal of the experiment is to evaluate the efficiency of our approach for the generation of the refactorings' sequences. In particular the experiment aimed at answering the following research questions:

RQ1: To what extent can the interactive approach generate correct refactorings' sequences?
RQ2: What types of refactorings are correctly suggested?

To answer these questions we implemented and tested the approach on open source projects. In particular, to answer RQ1, we used an existing corpus of known models refactorings to evaluate the precision and recall of our approach, and to answer RQ2, we investigated the type of refactorings that were suggested by our tool. In this section, we present the experimental setup and discuss the results of this experiment.

5.4.1 Supporting Tool and Experimental Setup

We implemented our approach as a plugin within the Eclipse™ development environment. Figure 5.4 shows a screenshot of the model refactoring plugin perspective. This plugin takes as input a base of examples of refactored models and an initial model to refactor. The user specifies the population size, the number of iterations, the individual size, the number of mutations, the number of interactions, and the percentage of the solutions shown in each interaction. It generates as output an optimal sequence of refactorings to be applied on the analyzed system.

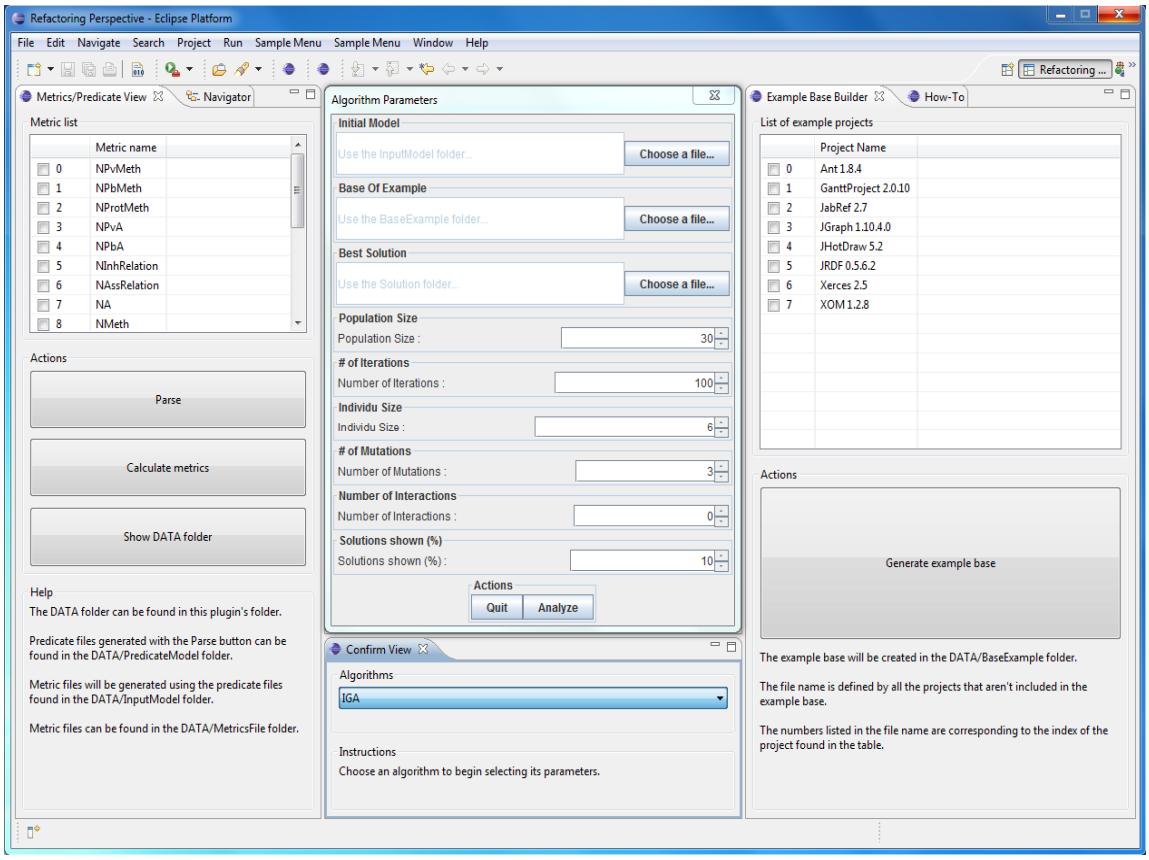


Figure 5.4 Model Refactoring Plugin

To build the base of examples, we used the Ref-Finder tool (Kim et al., 2010) to collect the refactoring that were applied on six Java open source projects (Ant, JabRef, JGraphx, JHotDraw, JRDF, and Xom). Ref-Finder helps retrieving the refactorings that a system has undergone by comparing different versions of the system. We manually validated the refactorings returned by Ref-finder before including them in the base of examples. To answer the research questions reported above, we analyzed two open-source Java projects in our experiment. We have chosen these open source projects because they are medium-sized open-source projects and they have been actively developed over the past 10 years. The participants in the experiment were three Ph.D students enrolled in Software Engineering and all of them are familiar with the two analyzed systems and have a strong background in object-oriented refactoring.

5.4.2 Results and discussions

To assess the accuracy of the approach, we compute the precision and recall of our IGA algorithm when applied to the two projects under analysis. In the context of our study, the precision denotes the fraction of correctly proposed refactorings among the set of all proposed refactorings. The recall indicates the fraction of correctly proposed refactorings among the set of all actually applied refactorings in the subsequent versions of the analyzed projects. To assess the validity of the proposed refactorings, we compare them to those returned by Ref-Finder when applied to the two projects and their subsequent versions. The precision and recall results might vary depending on the refactorings used, which are randomly generated, though guided by a meta-heuristic. Figure 5.5 and Figure 5.6 show the results of 23 executions of our approach on Xerces and GanttProject, respectively. Each of these figures displays the precision and the recall values for each execution.

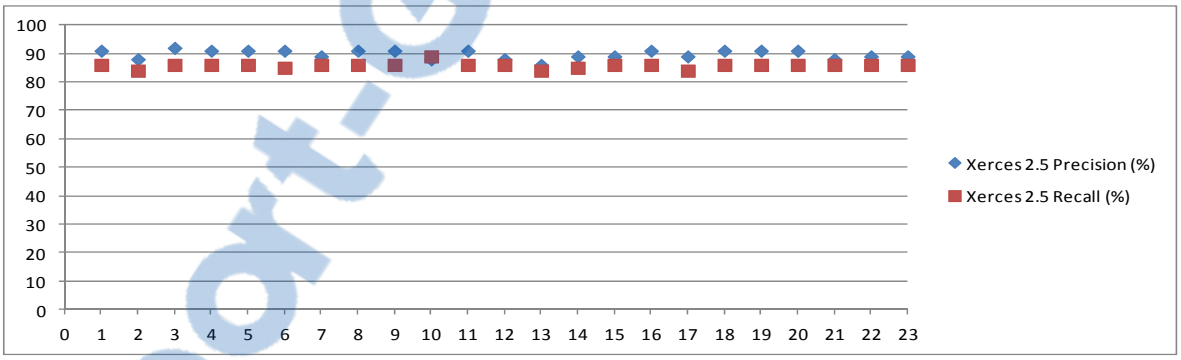


Figure 5.5 Multiple execution results for Xerces

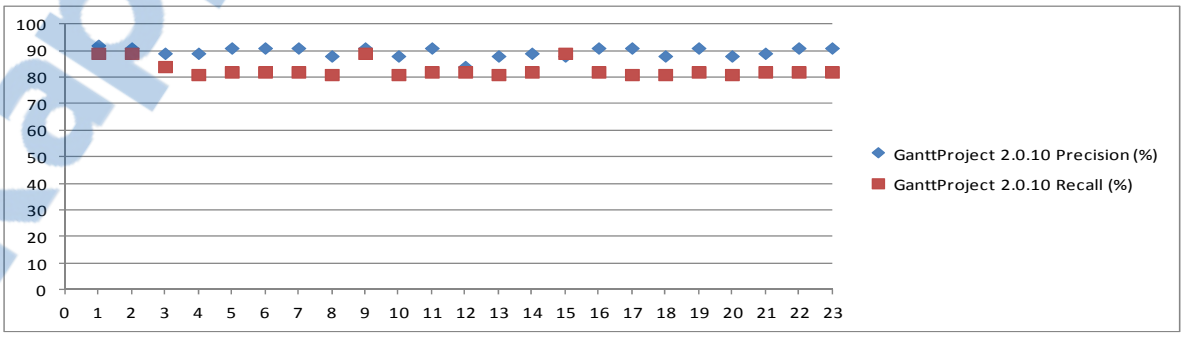


Figure 5.6 Multiple Execution results for GanttProject

Generally, the average precision and recall (87.8%) allows us to positively answer our first research question RQ1 and conclude that the results obtained by our approach are very encouraging. The precision in the two projects under analysis (on average 90% of all executions) proves that a big number of the refactorings proposed by our approach were indeed applied to the system's model in its subsequent version (i.e., the proposed refactorings match, in most cases, those returned by Ref-Finder when applied on the system's model and its subsequent version). To ensure that our results are relatively stable, we compared the results of the multiple executions (23) of the approach on the two analyzed projects shown in Figure 5.5 and Figure 5.6. The precision and recall scores are approximately the same for different executions in the two considered projects. We also compared the sequences of refactorings returned by different executions of our algorithm on the same project. We found that when a class (from the model under analysis) is part of two different returned sequences, the refactoring operations proposed for this class within these sequences are similar. We consequently conclude that our approach is stable.

Our experiment through the interactions with designers allowed us to answer the second research question RQ2 by inferring the types of refactorings they recognized as good refactorings. Figure 5.7 shows that 82% of the the *Move_method* and *Pull_up_method* refactorings proposed during the executions are recognized as good refactoring versus only 70% of the *Rename_method* refactorings. We noticed also, that only 9 of 12 refactorings used in the approach are considered in this analysis. This may result from the quality of the base of examples or from the random factor which characterizes genetic algorithm. We made a further analysis to understand the causes of such results. We found out that through the interactions, the designers have to recognize the meaningless refactorings and penalize them by assigning them a 0 as a rating value; this has significantly reduced the number of these types of refactorings in the optimal solution.

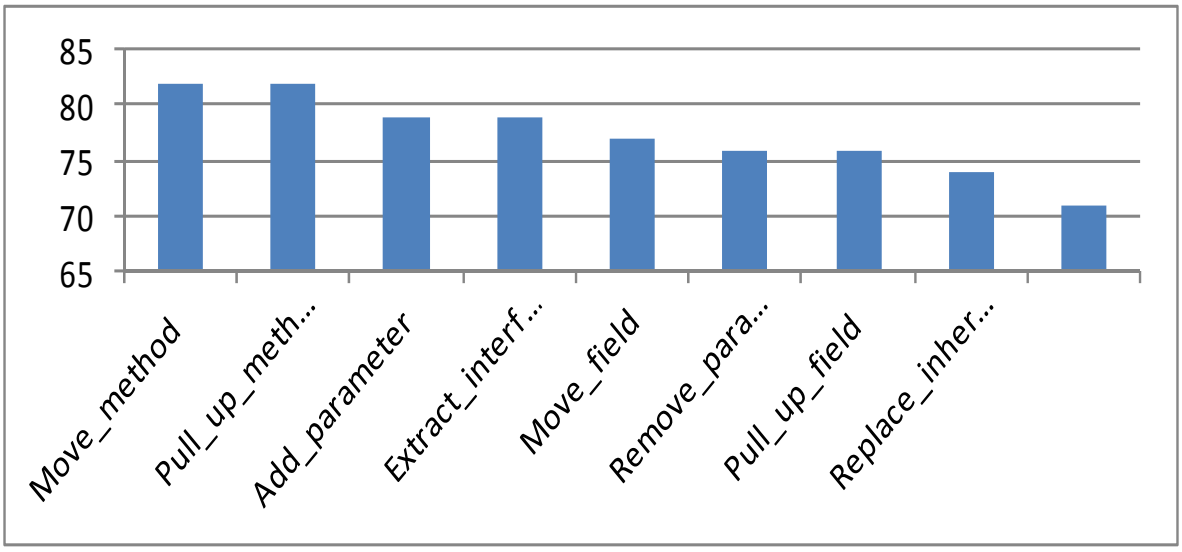


Figure 5.7 Distribution of refactorings recognized as correct refactorings through intercations

Despite the good results, we noticed a very slight decrease in recall versus precision in the analyzed projects. Our analysis pointed out towards two factors. The first factor is the project domain. In this study we tried to propose refactorings using a base of examples which contains different projects from different domains. We noticed that some projects focus on some types of refactorings compared to others (i.e., some projects in the base of examples has a big frequency of «*pull_up_Attribute*» and «*pull_up_method*»). The second factor is the number and types of refactorings considered in this experimentation. Indeed, we noticed that some refactorings (e.g., «*pull_up_method*», «*pull_up_Attribute*», «*add_parameter*») are located correctly in our approach. We have no certainty that these factors can improve the results but we consider analyzing them as a future work to further clarify many issues.

5.4.3 Threats to Validity

We have some points that we consider as threats to the generalization of our approach. The most important one is the use of the Ref_finder Tool to build the base of examples and at the same time we compare the results obtained by our algorithm to those given by Ref_finder. Other threats can be related to the IGAs parameters setting and to the use of Ph.D students in

the experiment to get feedbacks. Although we applied the approach on two systems, further experimentation is needed. Also, the reliability of the proposed approach requires an example set of applied refactoring on different systems. It can be argued that constituting such a set might require more work than these examples. In our study, we showed that by using some open source projects, the approach can be used out of the box and will produce good refactoring results for the studied systems. In an industrial setting, we could expect a company to start with some few open source projects, and gradually enrich its refactoring examples to include context-specific data. This is essential if we consider that different languages and software infrastructures have different best/worst practices. Finally, since we viewed the model refactorings' generation problem as a combinatorial problem addressed with heuristic search, it is important to contrast the results with the execution time. We executed the plugin on a standard desktop computer (i7 CPU running at 2.67 GHz with 8GB of RAM). The number of interactions was set to 50. The execution time for refactorings' generation with a number of iterations (stopping criteria) fixed to 1000 was less than seventy minutes. This indicates that our approach is reasonably scalable from the performance standpoint.

5.5 Conclusion and Future Work

In this article, we presented a new approach that aims to suggest appropriate sequences of refactorings that can be applied on a given design model and in particular on a UML class diagram. To do so, we adapted Interactive Genetic Algorithms (IGAs) to build an algorithm which exploits both existing model refactoring examples and the designer's knowledge during the search process for opportunities of model refactorings. We implemented the approach as a plugin integrated within the Eclipse platform and we performed multiple executions of the approach on two open source projects. The results of our experiment have shown that the approach is stable regarding its correctness, completeness and the type and number of the proposed refactorings per class. IGA has significantly reduced the number of meaningless refactorings in the optimal solutions for these executions. While the results of the approach are very promising, we plan to extend it in different ways. One issue that we

want to address as a future work is related to the base of examples. In the future we want to extend our base of examples to include more refactoring operations. We also want to study and analyze the impact of using domain-specific examples on the quality of the proposed sequences of refactorings. Actually, we kept the random aspect that characterizes genetic algorithms even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to correct the one under analysis. Finally, we want to apply the approach on other open source projects and further analyze the type of refactorings that are correctly suggested.

We noticed that our MOREX+I approach is capable to suggest most of the expected refactorings. Despite the designers' feedbacks, not all suggested refactorings are semantically meaningful. In addition, the MOREX+I approach is time-consuming and is a semi-automatic approach. In order to keep the automatic aspect and to avoid the time-consuming problem, we had the idea to introduce the semantic aspect as a second objective within a multi objective perspective. The next chapter details our multi-objective approach that consists of suggesting refactoring based on the calculation of both structural and semantic similarities by adapting the Non-dominated Sorting Genetic algorithm (NSGA-II) (Deb et al., 2002).

CHAPTER 6

EXAMPLE-BASED MODEL REFACTORING USING MULTI OBJECTIVE OPTIMIZATION

Adnane Ghannem, Ghizlane El Boussaidi[1] and Marouane Kessentini[2]

[1]Department of Software and IT Engineering, École de Technologie Supérieure,

1100 Notre-Dame West, Montreal, Quebec, (H3C 1K3) Canada

[2]Department of Computer and Information Science, University of Michigan - Dearborn

4901 Evergreen Road, Dearborn, MI 48128 USA

This paper has been submitted for publication in Journal of

Automated Software Engineering

ABSTRACT

Refactoring remains the most widely used technique for improving software quality. Most of the contributions in model refactoring were based on declarative rules to detect refactoring opportunities and to apply the appropriate refactorings. However, a high number of rules is required to obtain a complete specification of refactoring opportunities. In some situations, examples of refactorings from past maintenance experiences can be collected. Based on these observations, we considered the model refactoring problem as multi objective problem by suggesting refactorings sequences that should maximize both structural and semantic (syntactic) similarity between a given model (i.e., the model to be refactored) and a set of models in the base of examples (i.e., models that have undergone some refactorings). To this end, we use the Non-dominated Sorting Genetic Algorithm (NSGA-II) to find a set of representative Pareto optimal solutions that present the best trade-off between structural and semantic/syntactic similarities of models. The validation results on three systems of real world models taken from open-source projects and the comparison of our approach with two existing approaches confirm the effectiveness of our approach.

Keywords: software maintenance; model evolution; refactoring by example; NSGA-II; Pareto front.

6.1 Introduction

According to the ISO/IEC 14764 standard, the maintenance process includes the necessary tasks to modify existing software while preserving its integrity (ISO/IEC, 2006). Maintenance tasks can be seen as incremental modifications to a software system that aim to add or adjust some functionality or to correct some design flaws and fix some bugs. These software maintenance activities become more complex when the size of the system and the number of requirements increase during the time (Fowler, 1999). Therefore, it is important to provide automated and semi-automated software maintenance tools to improve the quality of software.

To meet software quality standards, developers need to continuously restructure the software system to improve its structure and design. This process is commonly called refactoring (Mens and Tourwé, 2004). According to Fowler (Fowler, 1999), refactoring is the disciplined process of cleaning up code to improve the software structure while preserving its external behavior. The process of refactoring involves several activities (Mens and Tourwé, 2004) including the activities of identifying refactoring opportunities in a given software and determining which refactorings to apply. Many researchers have been working on providing support for refactoring (e.g., (Opdyke, 1992), (Fowler, 1999), and (Moha, 2008)). However, they have, for the most part, focused on refactorings at the source code level (e.g., code smells (Du Bois et al., 2004)). Very few approaches tackled the refactoring process at the model level (e.g., (El-Boussaidi and Mili, 2011), (Mens et al., 2007a) and (Zhang et al., 2005)). Nevertheless, models are primary artifacts within the model-driven engineering (MDE) approach which has emerged as a promising approach to manage software systems' complexity and specify domain concepts effectively (Douglas, 2006). In MDE, abstract models are refined and successively transformed into more concrete models including executable source code. In this context, refactoring is a specific type of model transformation that aims at improving the quality of a given model; for example improving the design of an existing design model by applying a design pattern which can be encoded as a model transformation (El-Boussaidi and Mili, 2008).

Actually, the rise of MDE increased the interest and the needs for tools supporting refactoring at the model-level. However there are many open and challenging issues that we must address when building such a tool. Some of these challenges were identified in (Mens et al., 2007a) and they include issues related to assessing model quality, ensuring synchronization and coherence between models (including source code), preserving behavior, etc. Mens and Tourwé (Mens et al., 2007a) argue that most of the refactoring tools offer a semi-automatic support because part of the necessary knowledge for performing the refactoring remains implicit in designers' heads. Indeed, recognizing opportunities of model refactoring remains a challenging issue that is related to the model marking process within the context of MDE which is a notoriously difficult problem that requires design knowledge and expertise (El-Boussaidi and Mili, 2008). In addition, existing work on refactoring relies on declarative rules to detect and correct defects (i.e., refactoring opportunities) and the number of types of these defects can be very large (Kessentini et al., 2011b). Finally an issue that is common to most of refactoring approaches is the problem of sequencing and composing refactoring rules. This problem is related to the control of rules' applications within a rule-based transformational approach in general.

To overcome some of these issues, many approaches to refactoring are using a search-based approach where the refactoring is considered as an optimization problem (e.g. (O'Keeffe and O'Cinneide, 2006) (Seng et al., 2006) (Kessentini et al., 2008) (Ghannem et al., 2014b) (Ouni et al., 2013) and (Harman and Tratt, 2007)). Search-based refactoring approaches adapted some of the known heuristics methods such as Simulated annealing and Hill_climbing as proposed in (O'Keeffe and O'Cinneide, 2006) and (Seng et al., 2006), and Genetic Algorithms as proposed in (Kessentini et al., 2008). In previous work (Ghannem et al., 2014b), we proposed a by example approach that recommends refactorings to correct models. The approach uses single-objective optimization to find the best refactorings sequences that maximize the structural similarity between the model under analysis and a set of model refactoring examples. The structural similarity is computed using a set of metrics. Other optimization goals were considered in search-based refactoring approaches (e.g., reducing the refactoring effort (Ouni et al., 2013), improving the software structure (Seng et al.,

2006)). Harman and Tratt (Harman and Tratt, 2007) have proposed a multi-objective approach that uses two software metrics (CBO: coupling between objects, and SDMPC: standard deviation of methods per class) to define two optimization objectives. Most of these approaches relied on the structural information (i.e., a combination of software metrics) to formulate their fitness functions and do not consider semantics in the optimization process. However, to suggest meaningful refactorings and to reduce the number of possible refactorings, both quality and semantics of the model to be refactored should be considered.

In this paper, we propose a multi-objective optimization approach to find the best sequence of refactorings that maximizes both the structural and the semantic (syntactic similarities between names) similarity between a given model (i.e., the model to be refactored) and a set of models in the base of examples (i.e., models that have undergone some refactorings). We hypothesize that the knowledge required to propose appropriate refactorings for a given object-oriented model may be inferred from other existing models' refactorings when there are some semantic and structural similarities between these models and the given model. To this end, we adapt the Non-dominated Sorting Genetic Algorithm (NSGA-II) (Deb et al., 2002) which aims at finding a set of representative Pareto optimal solutions in a single run. Our approach takes as input an initial model which we want to refactor, a base of examples of models and their subsequent refactorings, and a list of metrics and semantic measures calculated on both the initial model and the models in the base of examples and it generates as output a solution to the refactoring problem. A solution consists of a list of refactoring operations that should be applied to the initial model. The process of generating this solution can be viewed as the mechanism that finds a list of refactoring operations with the best trade-off between the two criteria: structural and semantic similarities.

The primary contributions of the paper can be summarised as follows:

1. We introduce a novel multi-objective refactoring approach based on the use of examples. This approach relieves the designer from explicitly defining rules that detect opportunities of refactoring and that suggest the appropriate refactorings.

2. We take into consideration the semantics when comparing between the model to be refactored and existing model examples to suggest refactoring solutions.
3. We present and discuss the results of experiments with our approach and we compare these results to those of single objective approaches that do not consider model semantics.

The rest of this paper is organized as follows. Section 6.2 presents the overall approach and the details of our adaptation of the multi-objective evolutionary algorithm NSGA-II to the model refactoring problem. Section 6.3 describes the supporting tools and experimental settings and presents results and discussion. Related works are discussed in section 6.4 and we conclude and outline some future directions to our work in section 6.5.

6.2 Model Refactoring using multi Objective optimization

6.2.1 Approach Overview

The approach proposed in this paper exploits examples of model refactorings and an evolutionary algorithm (NSGA- II (Deb et al., 2002)) to automatically suggest sequences of refactorings that can be applied on a given model. The general structure of our approach is introduced in Figure 6.1. It takes as inputs a set of refactoring examples (label A) (i.e., existing models and their related refactorings), an initial model (label B) and takes as controlling parameters a set of software metrics (label C). The approach generates as output a sequence of refactorings that can be applied to the initial model. The process of generating a sequence of refactorings (Figure 6.1) can be viewed as the mechanism that finds the best way to select and combine refactoring operations among the ones in the base of examples, in such a way to maximize the structural and the semantic similarities between entities to be refactored in the initial model and entities of the models (from the base of examples) that have undergone the refactoring operations composing the sequence. The structural similarity between two entities (e.g., classes) is computed using software metrics of these entities while their semantic similarity is computed using semantic measures based on WordNet (Howe, 2009).

In our approach, we consider a subset of the 72 refactorings defined in (Fowler, 1999); i.e., only those refactorings that can be applied to UML class diagrams. Indeed, some of the refactorings in (Fowler, 1999) may be applied on design models (e.g. *Move_Method, Rename_method, Move_Attribute, Extract_Class* etc.) while others cannot be (e.g. *Extract_Method, Inline_Method, Replace_Temp_With_Query* etc.). In our approach we considered a list of twelve refactorings (e.g. *Extract_class, Push_down_method, Pull_up_method,* etc.). The choice of these refactorings was mainly based on two factors: 1) they apply at the class diagram level; and 2) they can be linked to a set of metrics (i.e., metrics which are impacted when applying these refactorings). In the context of our approach, we used a list of sixteen metrics that apply to class diagrams (e.g. *Number of attributes: NA, Number of methods: NMeth, Number of dependencies: NDep*, etc.). These metrics include the eleven metrics defined in (Genero et al., 2002) to which we have added a set of simple metrics (e.g., *number of private methods in a class, number of public methods in a class*). All these metrics are related to the class entity which is the main entity in a class diagram. These metrics are used to compute the structural similarities between classes from the initial model and those in the base of examples. To compute the semantic similarity between two classes, we use the Rita toolkit (Howe, 2009).

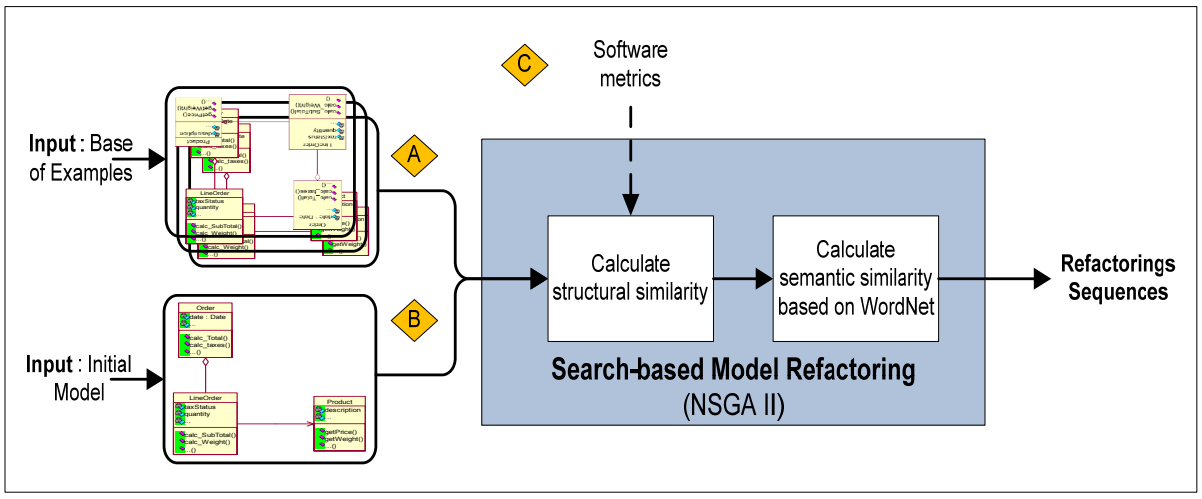


Figure 6.1 Multi-objective model refactoring using examples

To find the best trade-off between the two objectives (structural and semantic measures), we adapted the non-dominated sorting genetic algorithm (NSGA-II) (Deb et al., 2002). This algorithm and its adaptation to the refactoring problem are described in the next section.

6.2.2 NSGA-II for Model refactoring

6.2.2.1 NSGA-II overview

NSGA-II is an evolutionary algorithm that uses non-dominated sorting to solve multi-objective optimization problems (Deb et al., 2002). NSGA-II was designed to be applied to an exhaustive list of candidate solutions, which creates a large search space. The main idea of the NSGA-II is to find a representative set of Pareto optimal solutions, called non-dominated solutions. A solution is called non-dominated when no other solution can improve some optimization objective without degrading another. Given a set of objectives f_i, i$\in$ $1, \dots n$, to maximize, a solution x is said to Pareto dominate another solution x' if and only if:

$$\forall_i, f_i(x') \leq f_i(x) \ and \ \exists j | f_j(x') < f_j(x) \tag{6.1}$$

Three main steps characterize the NSGA-II algorithm:

1. Create randomly the initial population P_0 of individuals encoded using a specific representation.
2. Create a child population C_0 generated from the population of parents P_0 using genetic operators such as crossover and mutation.
3. Merge both populations and select a subset of individuals, based on the dominance principle to create the next generation.

This process is repeated until reaching the last iteration according to stopping criteria.

6.2.2.2 NSGA-II adaptation

We describe in this section how we adapted the NSGA-II to find the best trade-off between structural and semantic similarity. As our aim is to maximise both the structural and the semantic similarities, we consider each one of these criteria as a separate objective for NSGA-II. The pseudo-code for the algorithm is given inAlgorithm 6.1. The algorithm takes as input a set of model refactorings' examples (our base of examples), an initial model and set of metrics. Lines 1-2 construct an initial population which is a set of individuals that stand for possible solutions representing sequences of refactorings that can be applied to the classes of the initial model. An individual is a set of blocks where each block contains a class CIM (chosen from the initial model), a class CBE (from the base of examples) that was matched to CIM, and a list of refactorings which is a subset of the refactorings that were applied to CBE (in its subsequent version) and that can be applied to CIM. Individuals' representation is explained and illustrated in the following section.

After generating a population of refactoring solutions, the main NSGA-II loop (Lines 4-21) goal is to make a population of candidate solutions evolve toward the best sequence of refactoring, *i.e.,* an individual that maximises as much as possible both the semantic and the structural similarities between the classes CIM and CBE that were matched within the individual's blocks. During each iteration t, an offspring population C_t is generated from a parent population P_t using genetic operators (selection, crossover and mutation) (Line 5). Then, C_t and P_t are assembled in order to create a global population G_t. Then, each solution I in the population G_t is evaluated using our two fitness functions: (1) *structural function* to maximize (line 8): represent the structural similarity between CIM and CBE based on software metrics, (2) *semantic function* to maximise (line 9): calculates the semantics similarity between CIM and CBE using the semantic measures defined in Rita toolkit (Howe, 2009).

Once these functions are calculated, all the solutions will be sorted in order to return a list of non-dominated fronts F (F_1, F_2, ...), where F_1 is the set of non-dominated solutions, F_2 is the

set of solutions dominated only by solutions in F_1, etc (line 11). Then, we build the next population P_{t+1} from the set of non-dominated fronts starting from front F_1 to F_i (lines 14-17). In general, the number of solutions in all sets from front F_1 to Fi is larger than the Max_size. To choose exactly Max_Size solutions, we sort the solutions of the front Fi using the crowded-comparison operator ($<_n$) defined in (Deb et al., 2002) (line 18). Then, we select the best solutions needed until we reach the Max-size (line 19). The crowded-comparison operator ($<_n$) is based on non-domination ranking and the crowding distance described in (Deb et al., 2002). The algorithm terminates (line 21) when it achieves the termination criterion (i.e. maximum iteration number). The output of the algorithm is the set of best solutions, *i.e.*, those in the Pareto front of the last iteration (line 22). We give more details in the following sub-sections about the representation of solutions, genetic operators, and the fitness functions.

```
Inputs
Set of model refactorings' examples
Initial model
Set software metrics
Process
1     I := set (CIM, CBE, A set of applicable refactorings)
2     P0 := set_of(I)
3     t := 0
4     Repeat
5           Ct := apply_Genetic_Operators(Pt)
6           Gt := Pt  Ct
7           for all I  Gt
8                 Structural (I) := calculate_Structural_Similarity(CIM, CBE)
9                 Semantic(I) := calculate_Semantic_Similarity(CIM, CBE)
10          end for
11          F := fast_Non_Dominated_Sort (Gt)  // F = (F1, F2, ...)
12          Pt+1 := ϕ
13          i := 1
14          While |Pt+1| + |Fi|  Max_size
15                Pt+1 := Pt+1  Fi
16                i := i+1
17          end While
18          Sort (Fi , Crowded_Comparison_Operator)
19          Pt+1 := Pt+1  Fi [1 .. (Max_size - |Pt+1|)]
20          t := t+1
21    Until t = Max_iteration
22    best_solutions := first_front(Pt)
Output
best_solutions.
```

Algorithm 6.1 High-level code for NSGAII adaptation to our problem

1) Individual Representation

An individual is a set of blocks. A block contains three parts as shown by Figure 6.2: the first part contains the class CIM chosen from the initial model (model under analysis), the second part contains the class CBE from the base of examples that was matched to CIM, and finally the third part contains a list of refactorings which is a subset of the refactorings that were applied to CBE (in its subsequent versions) and that can be applied to CIM. Hence, the selection of the refactorings to be considered in a block is conformed to some constraints to

avoid conflicts and incoherence errors. For example, if we have a *Move_attribute* refactoring operation in the CBE class and the CIM class doesn't have any attribute, then this refactoring operation is discarded as we cannot apply it to the CIM class.

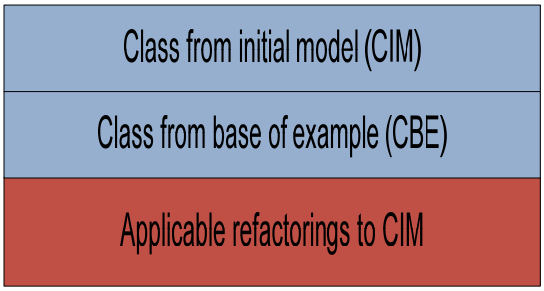


Figure 6.2 Block representation

The bottom part of Figure 6.3 shows an example of an individual (i.e., a candidate solution) composed of three blocks. Each block contains one refactoring operation. Hence the individual represents a sequence of refactoring operations to apply and the classes of the initial model on which they apply. The top part of Figure 6.3 shows the fragments of an initial model before and after the sequence of refactoring proposed by the individual (at the bottom of the figure) were applied. Notice that the same refactoring operation could be included several times in the same individual.

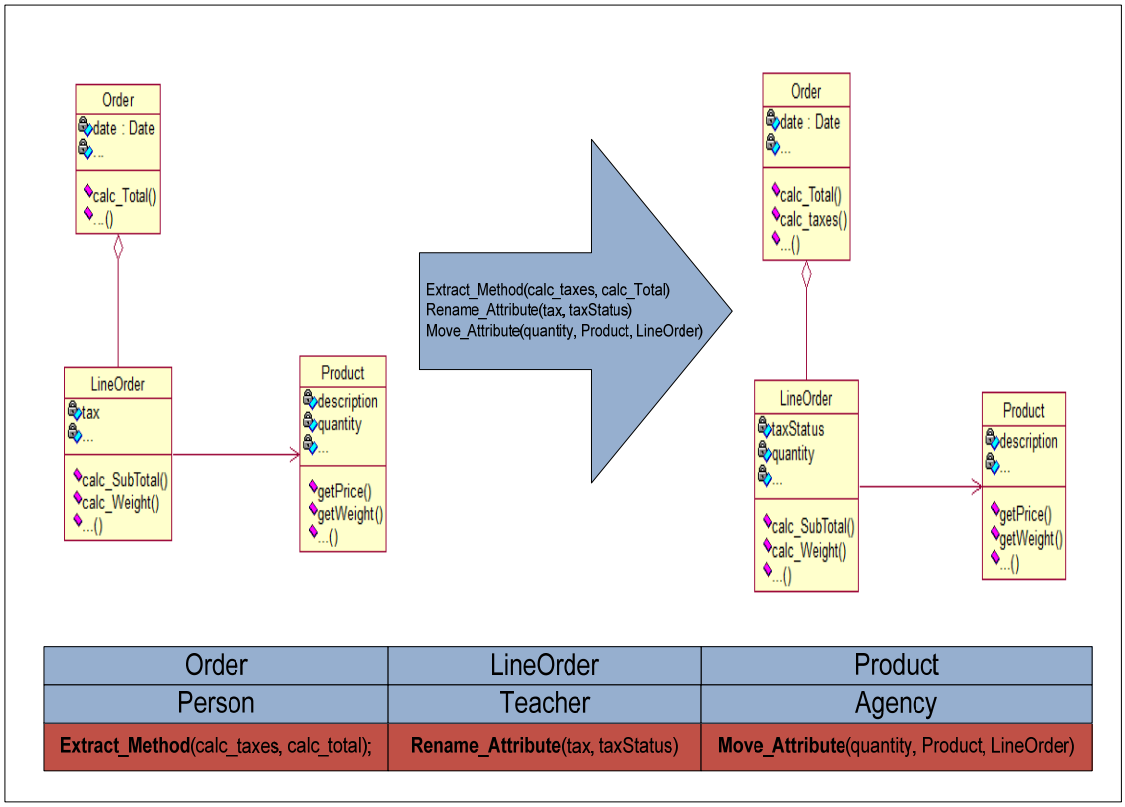


Figure 6.3 Individual representation

To generate an initial population, we start by defining the maximum individual size. This parameter can be specified either by the user or randomly. Thus, the individuals have different sizes. Then, for each individual we randomly assign: (1) A set of classes from the initial model that is under analysis and their matched classes from the base of examples, and (2) A set of refactorings that we can possibly apply on the initial model class among the refactorings proposed from the base of example class.

2) Selection and Genetic Operators

a) Selection

To select the individuals that will undergo the crossover and mutation operators, we used the binary tournament selection (Deb et al., 2002), which involves running several "tournaments"

among two individuals chosen at random from the population. Tournament selection also gives a chance to all individuals to be selected and thus it preserves diversity. At each iteration, two individuals chosen randomly are compared using the crowded-comparison-operator described in (Deb et al., 2002), i.e., the solution having better non-domination rank is preferred over the other, and in case of equal ranks, the solution having larger crowding distance is preferred over the other. We select half population as parents to perform crossover and mutation, and generate a full population of children.

b) Crossover

For each crossover, two individuals are selected randomly from the half population produced by the tournament selection. Even though individuals are selected, the crossover happens only with a certain probability. The crossover operator allows creating two offspring p'1 and p'2 from the two selected parents p1 and p2. It is defined as follows: A random position, k, is selected. The first k blocks of p1 become the first k blocks of p'2. Similarly, the first k blocks of p2 become the first k blocks of p'1. The rest of blocks (from position k+1 until the end of the sequence) in each parent p1 and p2 are kept. Figure 6.4 illustrates the crossover operator applied to two individuals (parents) p1 and p2. The position k takes the value 2 (number of blocks from left to right). The first two refactorings of p1 become the first two elements of p'2. Similarly, the first two refactorings of p2 become the first two refactorings of p'1.

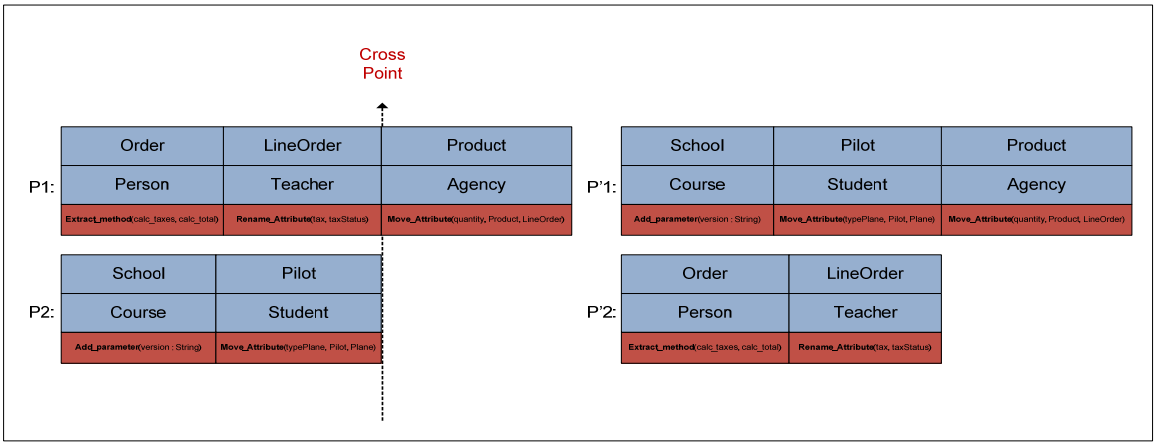


Figure 6.4 Crossover operator

c) Mutation

The mutation operator consists of randomly changing one or more blocks in the solution. Hence, given a selected individual, the mutation operator first randomly selects some blocks of the individual. Then, each selected block is modified by replacing its CBE class by another class randomly chosen from the base of examples. Figure 6.5 illustrates the effect of a mutation that replaced the refactoring *Rename_Attribute (tax, taxStatus)* applied to the class *LineOrder* (initial model) which was matched to the class *Teacher* (base of examples) by the refactoring *Rename_Method(calc_SubTotal, calc_TotalLine)* extracted from the new matched class *Student* (base of examples) and applied to the class *LineOrder* (initial model).

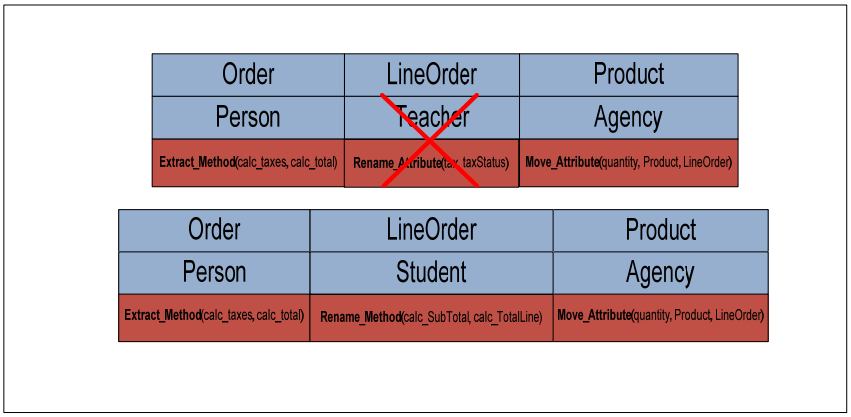


Figure 6.5 Mutation operator

3) Multi-criteria evaluation of individuals

Practically, the evaluation of an individual should be formalized as a mathematical function called “fitness function”. In this work, we considered two different fitness functions that, respectively, calculate structural and semantic similarities between classes in the initial model and classes in the base of examples. Our intuition is that a candidate solution that displays high structural and semantic similarities between the classes of the model and those chosen from the base of examples should give the best sequence of refactorings.

a) Structural criterion

The structural criterion is evaluated using the fitness function denoted by *Structural_Similarity* by formula 7.2 and 7.3.

$$Structural\_Similarity\ (CMI, CBE) = \frac{1}{m}\sum_{i=1}^{m} Sim(CMI_i, CBE_i) \tag{6.2}$$

$$Sim\ (CMI_i, CBE_i) = \begin{cases} 1 & if\ CMI_i = CBE_i \\ 0 & if\ (CMI_i = 0\ and\ CBE_i \neq 0\ )\ or\ (CMI_i \neq 0\ and\ CBE_i = 0) \\ \frac{CMI_i}{CBE_i} & if\ CMI_i < CBE_i \\ \frac{CBE_i}{CMI_i} & if\ CBE_i < CMI_i \end{cases} \tag{6.3}$$

Where ***m*** is the number of metrics considered in this project. CIM_i is the i^{th} metric value of the class CIM in the initial model while CBE_i is the i^{th} metric value of the class CBE in the base of examples. Using the similarity between classes, we define the structural fitness function of a solution, normalized in the range [0, 1], as:

$$f_{Structural} = \frac{1}{n}\sum_{j=1}^{n} Structural\_Similarity(CMI_{Bj}, CBE_{Bj}) \tag{6.4}$$

where ***n*** is the number of blocks in the solution and CMI_{Bj} and CBE_{Bj} are the classes composing the first two parts of the j^{th} block of the solution. To illustrate how the structural fitness function is computed, we consider a system containing two classes as shown in Table 6.1 and a base of examples containing two classes shown in Table 6.2. In this example we use six metrics and these metrics are given for each class in the model in Table 6.1 and each class of the base of examples in Table 6.2.

Table 6.1 Classes from the initial model and their metrics values

CMI	***NPvA***	***NPbA***	***NPbMeth***	***NPvMeth***	***NAss***	***NGen***
Plane	4	1	3	1	1	1
Product	2	2	6	0	1	0

Table 6.2 Classes from the base of examples and their metrics values

CBE	***NPvA***	***NPbA***	***NPbMeth***	***NPvMeth***	***NAss***	***NGen***
Student	2	1	3	0	3	0
Car	5	1	4	0	1	0

Consider an individual/solution I_1 composed by two blocks (*Plane*/*Student* and *Product*/*Car*). The structural fitness function of I_1 is calculated as follows:

$$f_{Structural\ I_1} = \frac{1}{2}\left[\frac{1}{6}\left[\left(\frac{2}{4}+1+1+0+\frac{1}{3}+0\right) + \left(\frac{2}{5}+\frac{1}{2}+\frac{4}{6}+1+1+1\right)\right]\right] = 0{,}41 \quad (6.5)$$

b) Semantic criterion

To formulate the semantic fitness function, we used Rita toolkit (Howe, 2009) which enables to compute the degree of likeness between two concepts based on their meaning. Specifically, we used this tool to calculate the semantic distance between two classes using their names. Thus, the semantic similarity between two classes Class1 and Class2, denoted as *Semantic_Similarity(Class1, Class2)*, corresponds to the semantic distance between *Class1' s name* and *Class2' s name*. Hence, the semantic fitness function of a solution corresponds to the average of semantic distances of all the blocks of the solution as shown by the formula 7.5.

$$f_{Semantic} = \frac{1}{n}\sum_{j=1}^{n} Semantic\_Similarity(CMI_{Bj}, CBE_{Bj}) \quad (6.6)$$

where ***n*** is the number of blocks in the solution and CMI_{Bj} and CBE_{Bj} are the classes composing the first two parts of the j^{th} block of the solution.

6.3 Experimentations with the approach

This section describes the evaluation steps of our approach. It starts by presenting our supporting tools. Then, we define our research questions. Finally, we describe our experimental settings and we present and discuss our results of the experimentations.

6.3.1 Supporting Tools

To validate our approach, we implemented a parser which analyses Java source code and generates a predicate model as illustrated in Figure 6.6. We used this parser to generate predicate models from 8 Java open source projects (Ant, JabRef, JGraphx, JHotDraw, GanttProject, JRDF, Xerces and Xom). To build the base of examples, we completed the generated models by manually entering the refactoring operations extracted with Ref-Finder (Kim et al., 2010), that these projects have undergone. An example of a class from the base of examples is shown in Figure 6.7. The Ref-Finder tool allows detection of complex refactorings (68 refactorings) which comprise a set of atomic refactorings by using logic-based rules executed by a logic programming engine. Ref-Finder helps finding refactorings that a system has undergone by comparing different versions of the system. We used the refactorings returned by Ref-finder for two raisons; build the base of examples and compute the precision and recall of our approach. To calculate the semantic similarity, we used the Rita toolkit (Howe, 2009).

```
Class (Class Name; visibiltiy)
{
        Attribute(Attribute Name; Type; visibility; ReturnType)
        ...
        Method(Method Name; [Parameters]; Visibility; Return Type;)
        ...
        Relation(Name of the source class; Name of the destination class;  Type of the relation)
        ...
}
```

Figure 6.6 Class representation in the generated model

```
Class (Class Name; visibiltiy)
{
        Attribute(Attribute Name; Type; visibility; ReturnType)
        ...
        Method(Method Name; [Parameters]; Visibility; Return Type;)
        ...
        Relation(Name of the source class; Name of the destination class;  Type of the relation)
        ...
        Refactoring(Refactoring Name (Parameters))
        ...
}
```

Figure 6.7 A class completed with its subsequent refactorings

We implemented our approach as a plugin within the Eclipse™ development environment. Figure 6.8 shows a screenshot of the model refactoring plugin perspective. The plugin supports many heuristic-based algorithms for refactoring and hence enable to enter many controlling parameters depending on the chosen algorithm. For the NSGA-II refactoring algorithm, it takes as input a base of examples of models and their related refactorings, an initial model to refactor, and a set of metrics. The user also specifies the population size, the number of iterations and the solution size (we also can keep this value randomly). It generates as output a Pareto front which contains optimal solutions of sequence of refactorings to be applied on the analyzed system.

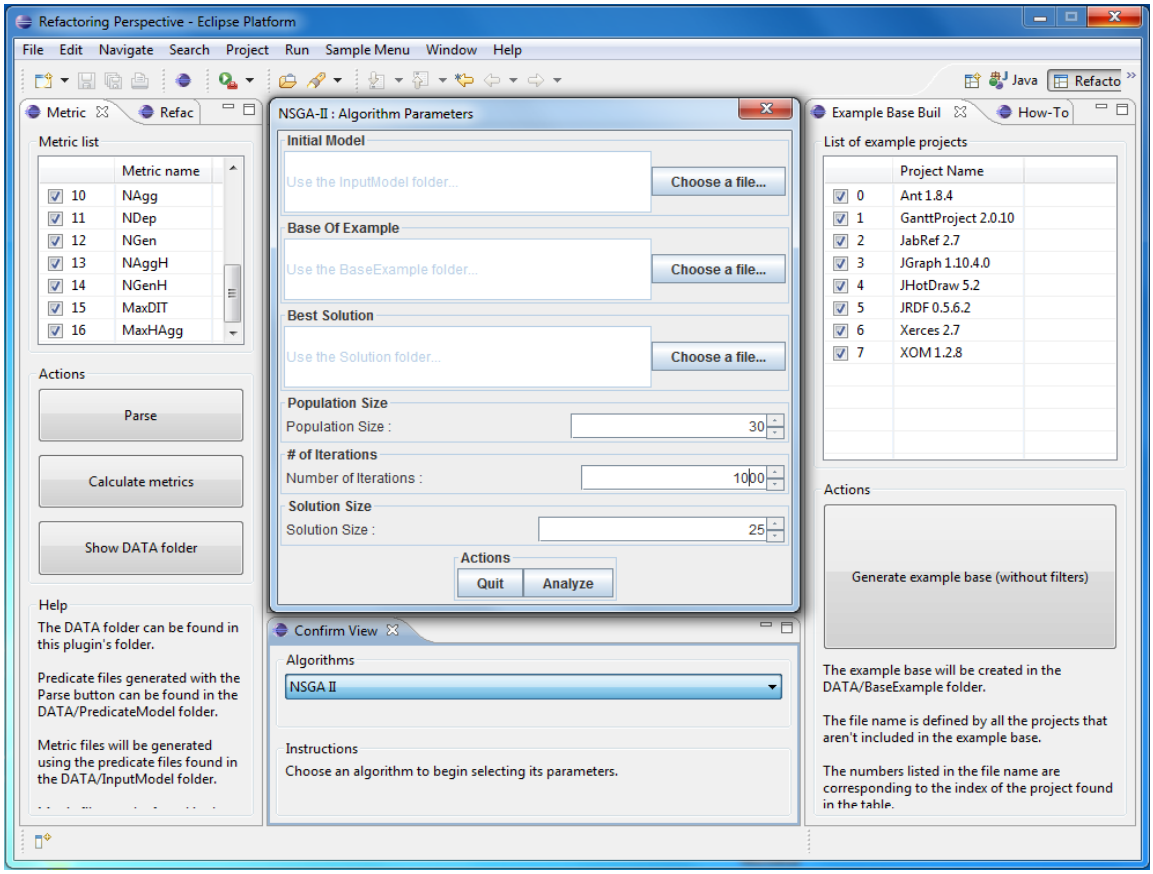


Figure 6.8 Model Refactoring Plugin

6.3.2 Research questions

The goal of our experimentation is to find out whether our approach could propose meaningful sequences of refactorings to correct design defects within models (e.g., class diagrams). Indeed, our approach addresses two research questions:

RQ1: To what extent can the proposed approach generate correct sequences of refactorings?

RQ2: To what extent can the semantics aspect improve the efficiency of our proposal to generate meaningful refactoring solutions?

To answer RQ1, we evaluated the precision and recall of our approach by applying it on a set of existing projects for which we had several versions and hence information about the refactorings they had undergone. To answer RQ2, we compared our results to those produced by our previous work called MOREX (Ghannem et al., 2014b) and GP (Kessentini et al., 2011a).

6.3.3 Experimental Setup

To set the parameters of NSGA-II for the search strategies, we performed several tests and the final parameters' values were set to a minimum of 1000 iterations for the stopping criterion and 30 as maximum size of population. We also set the crossover probability to 0.9 and the mutation probability to 0.4. These values were obtained by trial and error. We selected a high mutation rate because it allows the continuous diversification of the population, which discourages premature convergence to occur. We executed our algorithm on a standard desktop computer (i7 CPU running at 2.67 GHz with 8GB of RAM). The execution time for refactorings' generation with a number of iterations (stopping criteria) fixed to 1000 was less than 4 min. This indicates that our approach is reasonably scalable from the performance standpoint. However, the execution time depends on the number of refactorings and the size of the models in the base of examples.

To answer the research questions reported above, we analyzed three open-source Java projects in our experiment:

1) GanttProject (v 2.0.10): A Java project that supports project management and scheduling.
2) JHotDraw (v5.2): A framework for the creation of drawing editors.
3) Xerces (v2.7): A set of parsers compatible with Extensible Markup Language (XML).

We have chosen these open source projects because they are medium-sized open-source projects and most of them were analyzed in related work (e.g., (Kim et al., 2010), (Moha et

al., 2010), (Ghannem et al., 2011) and (Ouni et al., 2013)). Most of these open source projects have been actively developed over the past 10 years. Table 6.3 provides some relevant information about these projects.

Table 6.3 Case study settings

Model	**Classes**	**Methods**	**Attributes**	**Expected refactorings**
GanttProject 2.0.10	479	960	495	91
JHotDraw 5.2	160	519	141	71
Xerces 2.7	625	2113	1408	182

6.3.4 Results and discussion

Our results are presented based on the two indicators: precision and recall. For our validation we conducted multiple executions (31 executions) of our approach on GanttProject, JHotDraw and Xerces, respectively. Figure 6.9 illustrates the average of precision and recall values for each open source project over the 31 executions. Figure 6.10 shows an error bar plot displaying the average precision of these executions for each project. Similarly, Figure 6.11 shows an error bar plot displaying the average recall of these executions for each project.

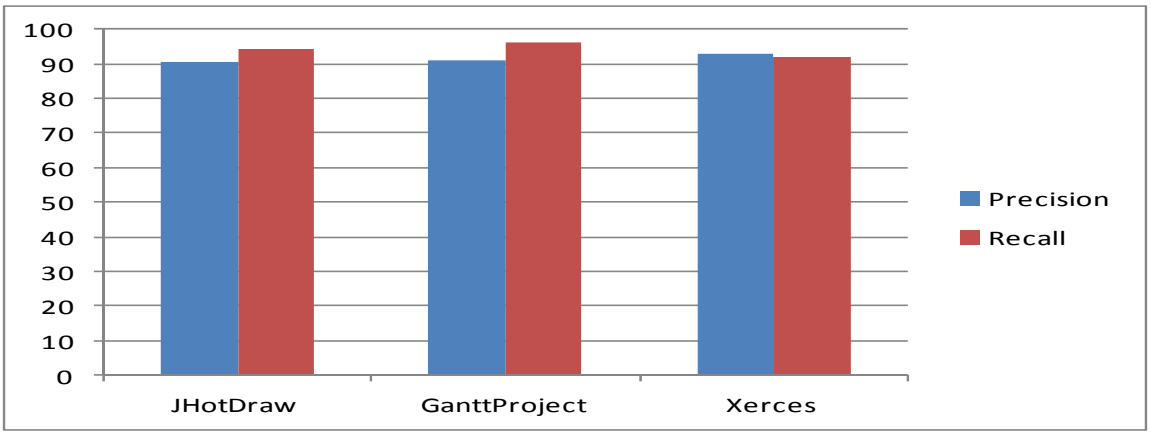


Figure 6.9 Average of precision and recall over 31 execution of our approach on JHotDraw, GanttProject and Xerces

We noticed very high values of the average of precision and recall which is over 90% for each project under test. Indeed, the confidence intervals for the precision and recall displayed by Figure 6.10 and Figure 6.11 respectively prove that precision and recall scores are approximately the same for different executions in the three projects under analysis. For JHotDraw, GanttProject and Xercess, the precision averages belong to these intervals [86.44%, 94.22%], [88.81%, 93.19%] and [89.62%, 96.42%] respectively with 95% of confidence. The recall averages belong to these intervals [91.18%, 97.42%], [92.40%, 99.6%], [89.03%, 94.65%] for JHotDraw, GanttProject and Xerces projet respectively with 95% of confidence. The small range between the lower and upper bounds within these intervals can be considered as sign of stability of the approach. These results allow us to positively answer our first research question RQ1 and conclude that the obtained results are very encouraging.

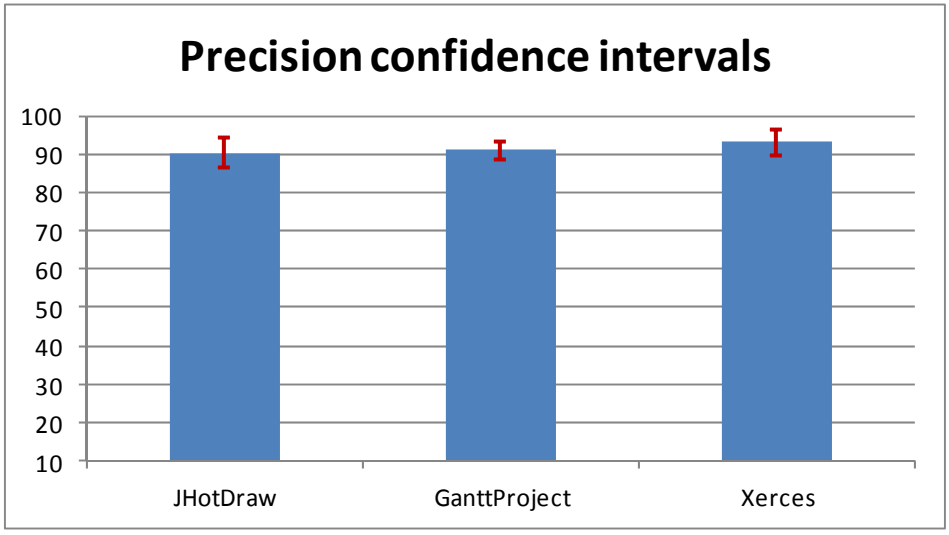


Figure 6.10 Confidences intervals for the precision average
of the 31 execusions on each project

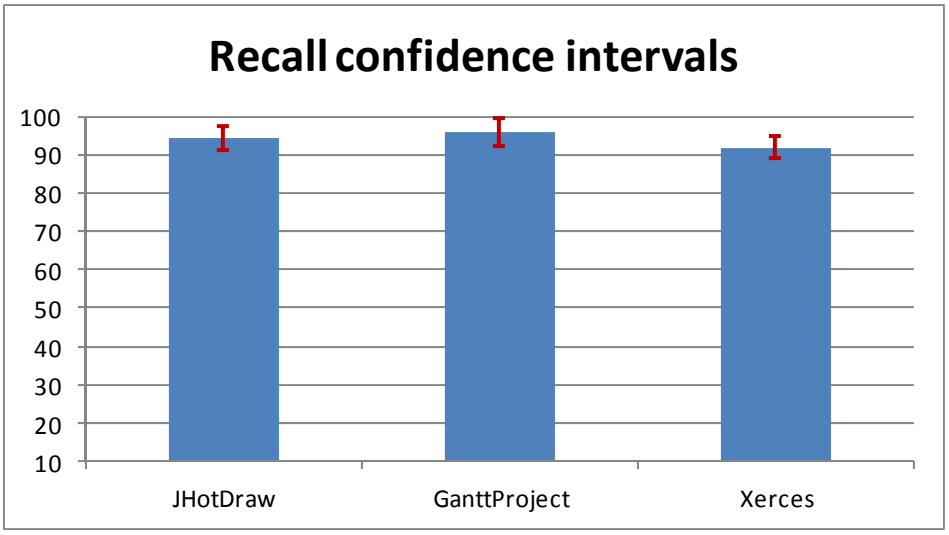


Figure 6.11 Confidence intervals for the recall average
of the 31 execusions on each project

The advantage with the multi-objective approach is that NSGA-II does not produce a single solution as GA, but a set of solutions called the Pareto front. In our context, NSGA-II converges to Pareto-optimal solutions that are considered as good trade-off between structural and semantic similarities. Figure 6.12, Figure 6.13 and Figure 6.14 display the pareto front for each of GanttProject, JHotDraw and Xerces projects, respectively. In these figures, each point is a solution with the structural similarity score represented in x-axis, the semantic similarity score in the y-axis (see ANNEX III, p. 217). The best solutions exist in the corner representing the Pareto-front that maximizes the values of the semantic and the structural similarities. The designer can choose a solution from this front depending on his preferences in terms of compromise.

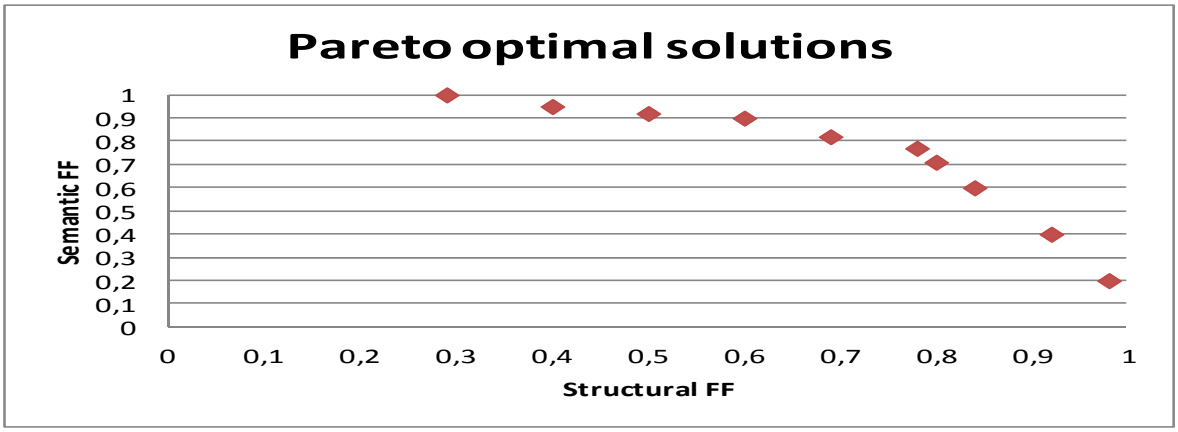


Figure 6.12 Pareto front for GanttProject 2.0.10

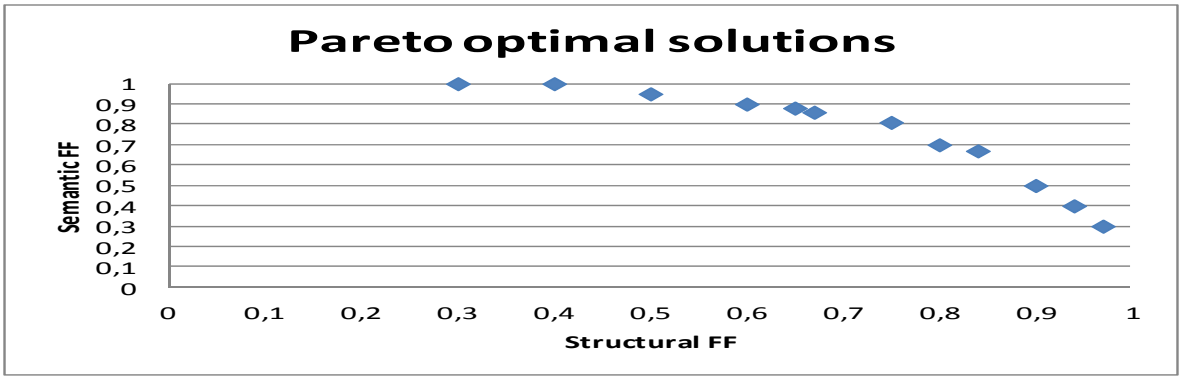


Figure 6.13 Pareto front for JHotDraw 5.2

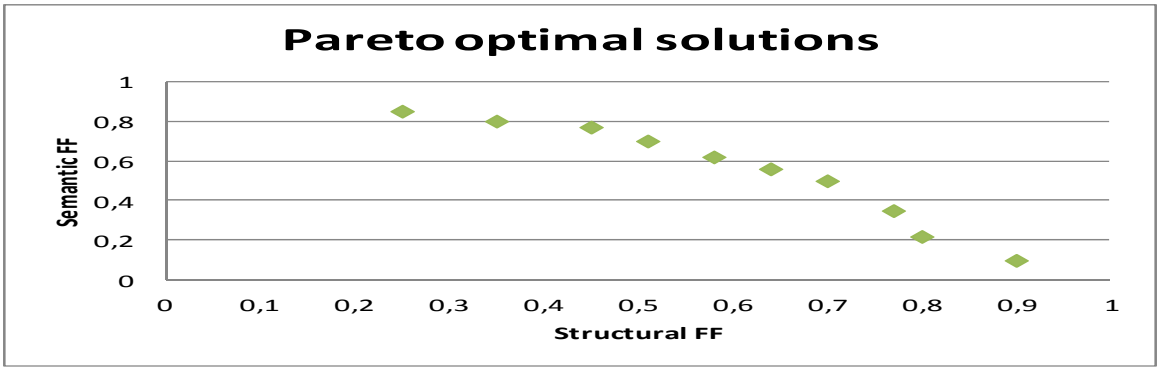


Figure 6.14 Pareto front for Xerces 2.7

To answer our second research question (RQ2), we compared the performance of our proposal (NSGA-II) to our previous work (MOREX: MOdel Refacttoring by EXamples)

(Ghannem et al., 2013), to a genetic programming (GP) based approach (Kessentini et al., 2011a) and to a random search approach. In a random search, the change operators (crossover and mutations) are not used, and populations are generated randomly and evaluated using the two fitness functions. In MOREX (Ghannem et al., 2014b), we used a single-objective genetic algorithm to propose refactorings; i.e., we considered only one fitness function based on the structural similarity which is a combination of software metrics. In (Kessentini et al., 2011a), the authors proposed an approach to generate detection rules based on quality metrics by using GP. As shown by Figure 6.15 and Figure 6.16, NSGA-II had higher average of precision and recall than MOREX and GP, and it beats by far the random search approach. For example for GanttProject, the NSGA-II average precision and average recall values are 91% and 96%, respectively, while these values are 82% and 86% in MOREX, 78% and 82% in GP, and 41% and 43% in the random search algorithm where these values do not exceed 50% for all the three projects. An increase of 9% on average between NSGA-II and MOREX is considered a great improvement. This improvement can be interpreted as the result of including the semantic similarity in addition to the structural similarity when evaluating the proposed refactoring solution.

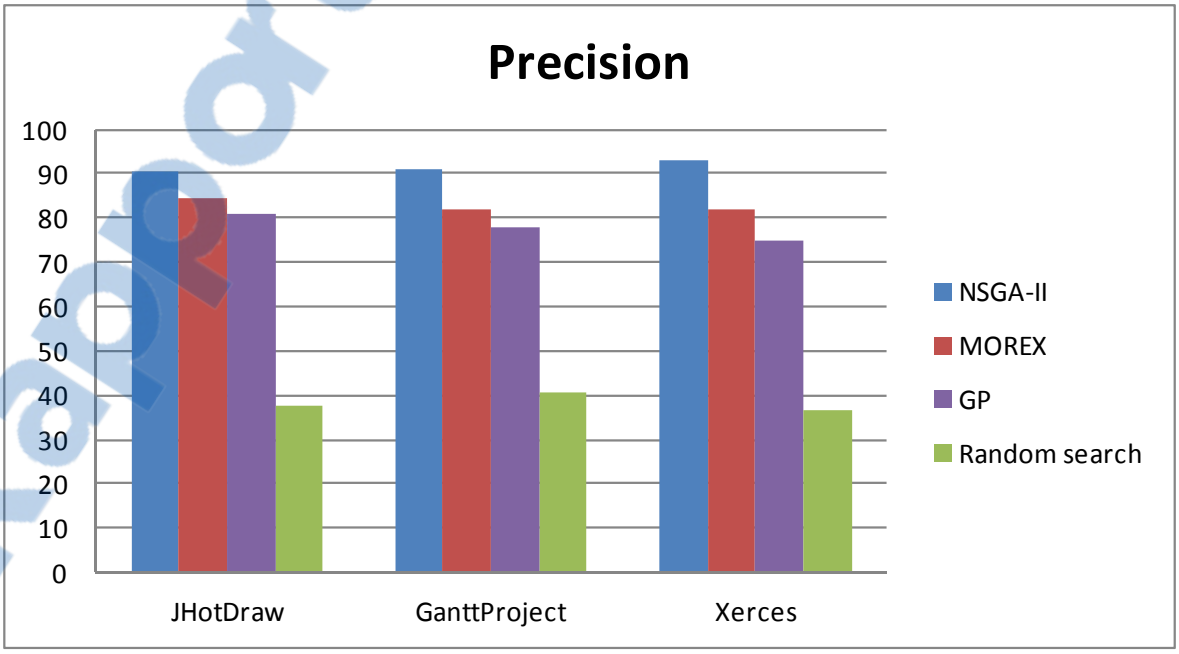


Figure 6.15 Comparison between NSGA-II, MOREX (Ghannem et al., 2013), GP (Kessentini et al., 2011a) and Random search in terms of precision

Figure 6.16 Comparison between NSGA-II, MOREX (Ghannem et al., 2013), GP (Kessentini et al., 2011a) and Random search in terms of recall

As the considered algorithms are meta-heuristics, they can produce different results on every run when applied to the same problem instance. For this reason, we used the p-values of the Wilcoxon rank-sum test (Wilcoxon, 1945) as a statistical test to compare the results of the three algorithms : NSGA-II, MOREX(Ghannem et al., 2014b) and GP (Kessentini et al., 2011a). We independently performed 31 executions using the algorithms in (Ghannem et al., 2014b) and (Kessentini et al., 2011a) for the three open-source projects that we used in our experiment. In our context, a p-value that is less than or equal to α (=0.05) means that the distributions of the results of the three algorithms are different in a statistically significant way. In fact, we computed the p-values of MOREX and GP results compared with NSGA-II. In this way, we could decide whether the outperformance of our approach over the MOREX and GP approach is statistically significant. The p-value for the precision median results of MOREX compared with NSGA-II is 0.0112 while the p-value of the recall median results of MOREX compared with NSGA-II is 0.0142. In addition, the p-value for the precision median results of GP compared with NSGA-II is 0.0098 while the p-value of the recall median results of GP compared with NSGA-II is 0.0087. Accordingly, we infer that the

precision and recall median values of our algorithm are statistically different from the MOREX and the GP ones on each of the systems based on the fact that these p-values are less than α (= 0.05). We consequently conclude that our approach is more effective than these two approaches and specifically it is more effective than an approach based only on the structural similarity without taking into account the semantics of the entities of analyzed models (i.e., MOREX). This observation allows us to positively answer our second research question RQ2.

6.4 Related Work

In this section, we summarize existing approaches where search-based techniques have been used to automate refactoring activities. We classify the related work into two main categories: single-objective and multi-objective optimization approaches.

In the first category, the majority of existing works combine several software metrics in a single fitness function to find the best sequence of refactorings (e.g. (Seng et al., 2006) (O'Keeffe, 2008) (Qayum and Heckel, 2009) (Kessentini et al., 2011a) and (Ghannem et al., 2013)). O'Keeffe et al. (O'Keeffe, 2008) present a comparative study of four heuristic search techniques applied to the refactoring problem. The fitness function used in this study was based on a set of 11 metrics to evaluate the quality improvements. The results of the experiments on five open-source systems showed that hill-climbing performs better than the other algorithms. Seng et al. (Seng et al., 2006) have proposed a single-objective optimization based on genetic algorithm (GA) to suggest a list of refactorings. The search process uses a single fitness function to maximize a weighted sum of several software metrics to improve the class structure of a system. Used metrics are related to some properties such as coupling, cohesion, complexity and stability. Indeed, the authors used some preconditions for each refactoring. These conditions are able to preserve the program behaviour (refactorings feasibility), but not the semantics domain. In addition, the validation was done only on the move method refactoring. Qayum et al. (Qayum and Heckel, 2009) have considered the problem of refactoring scheduling as a graph transformation problem.

They expressed refactorings as a search for an optimal path, using Ant colony optimization, in the graph where nodes and edges represent respectively refactoring candidates and dependencies between them. However, the use of graphs does not consider the domain semantics of the program and its runtime behavior. Kessentini et al. (Kessentini et al., 2011a) have proposed a single-objective combinatorial optimization using genetic algorithm to find the best sequence of refactoring operations that improve the quality of the code by minimizing as much as possible the number of design defects detected on the source code.

In the second category, Harman and Tratt (Harman and Tratt, 2007) have used the Pareto optimality concept to improve search-based refactoring approaches by combining two software metrics, CBO (coupling between objects) and SDMPC (standard deviation of methods per class), in two separate fitness functions. The multi-objective algorithm could find a good sequence of move method refactorings that should provide the best compromise between CBO and SDMPC to improve code quality. In (Ouni et al., 2013), the authors have proposed a multi-objective optimization approach to find the best sequence of refactorings using NSGA-II. This approach is based on two fitness functions: quality and effort. The quality corresponds to the number of corrected defects that are detected on the initial program, and the effort fitness function corresponds on the code modifications score. This approach recommends a sequence of refactorings that provide the best tradeoff between quality and effort. Ó Cinnéide et al. (O' Cinnéide et al., 2012) have proposed a search-based refactoring to conduct an empirical investigation to assess some structural metrics and to explore relationships between them. To this end, they have used a variety of search techniques (Pareto-optimal search, semi-random search) guided by a set of cohesion metrics. To conclude, the vast majority of existing search-based software engineering approaches focused only on the program structure improvements based on a set of software metrics in both single and multi-objectives approaches.

Other contributions which tackled the automating of refactoring activities at the model-level are based on rules that can be expressed as assertions (i.e., invariants, pre-and post-condition) (Ragnhild et al., 2007; Van Kempen et al., 2005), or graph transformations targeting

refactoring operations in general (e.g., (Biermann, 2010; Mens et al., 2007b)) or refactorings related to design patterns' applications (e.g., (El-Boussaidi and Mili, 2011)). The use of invariants (Ragnhild et al., 2007) has been proposed to detect some parts of the model that require refactoring. Refactorings are expressed using declarative rules. However, a complete specification of refactorings requires an important number of rules and the refactoring rules must be complete, consistent, non-redundant and correct. In (El-Boussaidi and Mili, 2011) refactoring rules are used to specify design patterns' applications. In this context, design problems solved by these patterns are represented using models and the refactoring rules transform these models according to the solutions proposed by the patterns. However, not all design problems are representable using models; i.e., for some patterns (e.g., Observer), the problem space is quite large and the problem cannot be captured in a single, or a handful of problem models (El-Boussaidi and Mili, 2011). Finally an issue that is common to most of these approaches is the problem of sequencing and composing refactoring rules. This is related to the control of rules' applications within rule-based transformational approaches in general.

6.5 Conclusion

We presented a by example search-based approach that exploits both structural and semantic information to improve the automation of suggesting refactoring. Our approach takes as input a model to be refactored, a base of examples of models and their subsequent refactorings, and a list of metrics and semantic measures calculated on both the initial model and the models in the base of examples. The output is a solution to the refactoring problem. A solution is a list of refactoring operations that should be applied to the initial model and that displays the best trade-off between the two criteria: structural and semantic similarities. In contrast to existing work on refactorings, semantics is a major concern in our paper.

Our experimentation shows that our technique outperforms state-of-the-art techniques where single-objective is used. The proposed approach has been evaluated on real-world models extracted from three open source systems. The experimental results indicate that the

proposed refactorings are comparable to those expected, i.e., the proposed refactorings match those returned by the Ref-Finder tool when applied on a model and its subsequent version. We also performed multiple executions of the approach on the three open source projects and the results have shown that the approach is stable regarding its precision and recall. These results allowed us to conclude that the proposed approach is more efficient and promising than approaches that do not consider the semantics in their optimization objectives.

In the future, we plan to extend our approach in different ways. One issue that we want to address as a future work is related to the base of examples; we want to extend it to include more refactoring operations. We also want to study and analyze the impact of using domain-specific examples on the quality of the proposed sequences of refactorings. Actually, we kept the random aspect that characterizes evolutionary algorithms even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to correct the one under analysis. We also plan to compare our results with other existing approaches other than the Ref-Finder tool and perform a further analysis on the nature and type of refactorings that are easier or harder to detect. We are also planning to add other significant objectives which can improve the quality of refactorings. To this end, we can explore some semantic properties other than the classes' names.

CONCLUSION

Developing tools that support software refactoring remains a challenge in software engineering. In this thesis, we focused on the problem of refactoring UML class diagrams as they are among the most used models in software design. In particular, we proposed approaches to detect design defects in class diagrams and to suggest appropriate refactorings to correct these defects. In this chapter, we summarise the contributions of our thesis and discuss some open issues and future directions of our work.

Contributions

The contributions of this thesis are approaches that support software designers and developers in the models' maintenance by automating the refactoring process of class diagrams. Contrary to traditional refactoring approaches that rely on declarative rules, our approaches exploit existing design defects examples and related refactorings to detect defects and suggest refactorings. Hence our contributions are divided into two categories: 1) those that support defects detection, and 2) those that support the correction of defects. These contributions are summarized in the following subsections.

Detecting design defects

Regarding the detection of design defects, we proposed two approaches. In the first one, we proposed a new detection mechanism for design defects detection. The approach derives rules in the form of metric/threshold combinations, from known instances of design defects (defect examples). Due to the large number of possible combinations, we used a GP as an heuristic that finds the best trade-off when combining metrics to build rules. Using GP, our approach has allowed automatic generation of rules to detect defects, thus relieving the designer from a fastidious manual rule definition task. We evaluated our approach by finding three potential design defect types in two large class diagrams (GanttPtoject and LOG4J). For

all these models, we succeeded in detecting the majority of expected defects. For example, for GanttProject, the average of our precision was 91%.

In the second approach, we proposed to identify design defects using GA based on the similarity/distance between the system under study and a set of defect examples without rule's definition. We tested our approach on four open-source systems (GanttProject, Log4J, ArgoUML and Xerces) to identify three potential design defects (blobs, functional decomposition and data class). Almost all the identified riskiest classes (the average of precision is around 95%) were found in a list of classes tagged as defects.

Suggesting refactorings

Regarding model refactoring, our contributions include refactoring mechanisms that do not require rule's definition. In this context, we proposed three approaches.

The first approach called MOREX consists in automatically suggesting model refactoring using GA. MOREX relies on a set of refactoring examples to propose sequences of refactorings that can be applied on a given class diagram. The refactoring is seen as an optimization problem where different sequences of refactorings are evaluated depending on the similarity between the model under analysis and the models in the examples. We evaluated MOREX on real-world models extracted from eight open source systems. The results indicated that the proposed refactorings are comparable to those expected. We also performed multiple executions of the approach on the 8 open source projects and the results have shown that the approach was stable regarding its precision and recall (average of 85%).

In the second approach, we adapted IGA to build an algorithm which exploits both existing model refactoring examples and the designer's knowledge during the search process for opportunities of model refactorings. We performed multiple executions of the approach on two open source projects (Xerces and GanttProject). IGA has significantly reduced the number of meaningless refactorings.

In the third contribution, we presented a by example search-based approach that exploits both structural and semantic information to improve the automation of suggesting refactoring. The solution in this approach was a list of refactoring operations that should be applied to the initial model and that displays the best trade-off between the two criteria: structural and semantic similarities between the analyzed model and the models in the base of examples. For the evaluation, we used real-world models extracted from three open source systems. The experimental results indicated that the proposed refactorings are comparable to those expected (JHotDraw, GattProject and Xerces) with a high precision and recall. These results allowed us to conclude that the approach which considers the semantic is more efficient and promising than approaches that do not consider the semantics in their optimization objectives.

All the proposed approaches, yield good quality solutions (i.e, appropriate refactoring sequences) in an acceptable execution time.

Limitations and Open Issues

While the results of the proposed approaches are very promising, there are some limitations and open issues that need to be addressed in the near future.

The first limitation to our work is related to the base of examples. Indeed, the evaluation of our approaches showed that the correctness of the proposed refactorings is closely linked to the quality of the base of examples. The availability and the quality of the base of examples could substantially affect the quality of the results of our approaches. This limitation is common to all example-based approaches. Moreover, the base of examples could be difficult to build. However, as we have shown in our experiments, only few examples are needed to obtain good results. In an industrial setting, we could expect a company to start with some few open source projects, and gradually migrate its set of refactoring examples to include context-specific data. This might be essential if we consider that different languages and software infrastructures have different best/worst practices.

The second limitation concerns the optimization process which could be time consuming for large models due to the nature of our solution. In addition, different executions for the same input, using evolutionary algorithms, could give different solutions (i.e., different sequence of refactorings). This observation can be a weakness for some model driven engineering applications where the output should be unique based on a deterministic process. However, two different solutions (i.e., sequences of refactorings) may be equivalent when their application to the model under study yields the same result. Obtaining equivalent solutions is the same as having different solutions given by different experts to resolve the same problem.

The third limitation that we consider as a threat to the generalization of our approach is a limitation related to the use of Ref_finder tool. First we built our base of examples using Ref_finder to extract the refactoring examples that different open source systems have undergone. Hence the examples we used are only those that Ref_finder is able to detect. Moreover, the most important issue is that we used Ref_finder to build the base of examples while at the same time we compared the results obtained by our algorithm to those given by Ref_finder. Regarding our dependence on the Ref_finder tool and the quality of the base of examples, we have already launched a project which consists in building a tool to extract refactorings from two subsequent versions of the same program. The tool relies on Abstract Syntactic Trees (AST) and it will cover most of the refactorings defined in (Fowler, 1999).

The last limitation concerns the results analysis. To validate our results, we made a quantitative analysis essentially based on precision and recall indicators. This kind of analysis enables us to make sense of the collected results summarised and presented as graphs. However, the quantitative analysis gives less detail than qualitative analysis that we plan to do it in the near future.

Future Research Directions

Various future work directions can be explored. First, we plan to extend our MOREX approach by considering two important factors. The first factor is the project domain. We

want to study and analyze the impact of using domain-specific examples on the quality of the proposed sequences of refactorings. Actually, in MOREX, we kept the random aspect that characterizes GAs even in the choice of the projects used in the base of examples without prioritizing one or more specific projects on others to correct the one under analysis. The second factor is the number and types of refactorings considered in our experimentations. We have no certainty that these factors can improve the results but we consider analyzing them as a future work to further clarify many issues.

We would also like to extend our third contribution (MOREX+I) about application of IGA within by-example model refactoring context. We want to expand the number of participants in the experiment and to work with many research teams. The idea is to collect the results from many experts in this area, i.e., model refactoring, who run our tool on the same inputs. The collected data may be useful to validate the approach on a large scale. Regarding our fourth contribution, i.e., multi objective optimization, we plan to extend it by adding other significant objectives which can improve the quality of refactorings. To this end, we could explore some semantic properties other than the classes' names (e.g., methods' and attributes' names).

In this thesis, we only looked at some activities that are part of the model refactoring process, namely defects detection and correction by suggesting appropriate refactorings. As part of future work, we plan to explore other activities such as the application of the refactoring and the behaviour preservation. In particular, we plan to build on the knowledge we acquired through our work on refactoring to propose approaches that may help in preventing the introduction of design defects in existing software. To do so, we will study the types of changes that software might have undergone through the analysis of its subsequent versions. The goal will be to classify these changes according to a degree of risk. A risky change can, for example, generate a design defect, a bug or a system malfunction, etc. To prevent the introduction of new defects in existing software, we aim to automatically identify the type of changes the designer is performing and notify him when the changes are prone to defects. In this context, we plan to explore and adapt existing classification algorithms (e.g., SVM).

ANNEX I

DESIGN DEFECTS IN GANTTPROJECT 2.0.10

N°	Classes	BLOB	FUNCTIONAL DECOMPOSITION	DATA CLASS
1	GanttGraphicArea Blob	√		
2	GanttOptions Blob	√		
3	GanttProject Blob	√		
4	GanttTree Blob	√		
5	ResourceLoadGraphicArea Blob	√		
6	TaskImpl Blob	√		
7	GanttTaskPropertiesBean Blob	√		
8	CSVSettingsPanel Blob	√		
9	ResourceAction SC	√		
10	CommonPanel SC	√		
11	ColorConvertion FD		√	
12	GregorianTimeUnitStack FD		√	
13	ColorValueParser FD		√	
14	DialogAligner FD		√	
15	GanttCalendarDays FD		√	
16	GanttPrintable FD		√	
17	PDFExportProcessor FD		√	
18	OpenFileDialog FD		√	
19	NewProjectWizard FD		√	
20	GanttTXTOpen FD		√	
21	AdjustTaskBoundsAlgorithm FD		√	
22	FindPossibleDependeesAlgorithm Impl FD		√	
23	RecalculateTaskCompletionPercentageAlgorithm		√	
24	MonthTextFormatter FD		√	
25	TaskHierarchyManagerImpl FD		√	
26	GraphicPrimitiveContainer FD		√	
27	GanttApplet FD		√	
28	GanttDialogInfo DataClass			√
29	CSVOptions DataClass			√
30	PrjInfos DataClass			√
31	GanttPaintParam DataClass			√
32	GanttURLChooser DataClass			√
33	GanttExportSettings DataClass			√
34	GanttDependStructure DataClass			√

35	ResourceEvent DataClass			√
36	ProjectResource DataClass			√
37	Notes DataClass			√

ANNEX II

OBTAINED RESULTS FOR GANTTPROJECT 2.0.10

Solution in the 1st execution	
Block 1	**CIM:** net.sourceforge.ganttproject.action.NewArtefactAction
	CBE: java.net.sf.jabref.HelpAction
	The obtained Sequence Of Refactoring (SOR): Pull_up_field(myIconOnMouseOver;net.sourceforge.ganttproject.action.NewArtefactAction; GPAction)
	The expected SOR: Pull_up_field(myIconOnMouseOver; NewArtefactAction; GPAction)
Block 2	**CIM:** net.sourceforge.ganttproject.action.ScrollGanttChartRightAction
	CBE: com.mxgraph.swing.handler.mxConnectionHandler
	The obtained SOR: Pull_up_method(getIconFilePrefix();net.sourceforge.ganttproject.action.ScrollGanttChartRightAction; GPAction)
	The expected SOR: Pull_up_method(isIconVisible; ScrollGanttChartRightAction; GPAction)
Block 3	**CIM:** net.sourceforge.ganttproject.chart.ChartGridImpl
	CBE: com.mxgraph.view.mxSpaceManager
	The obtained SOR: Pull_up_field(myCurrentFrame;net.sourceforge.ganttproject.chart.ChartGridImpl; ChartRendererBase) Pull_up_method(ChartGridImpl();net.sourceforge.ganttproject.chart.ChartGridImpl; ChartRendererBase)
	The expected SOR: Pull_up_field (areUnitsAccepted; ChartGridImpl; ChartRendererBase) Pull_up_method(getLineTopPosition; ChartGridImpl; ChartRendererBase)
Block 4	**CIM:** net.sourceforge.ganttproject.chart.TaskRendererImpl
	CBE: java.net.sf.jabref.SidePaneComponent
	The obtained SOR: Pull_up_field(myProvider;net.sourceforge.ganttproject.chart.TaskRendererImpl; GPAction) Pull_up_method(NewArtefactAction();net.sourceforge.ganttproject.chart.TaskRendererImpl; GPAction)
	The expected SOR: Pull_up_field(myIconOnMouseOver; NewArtefactAction; GPAction) Pull_up_method(isIconVisible; NewArtefactAction; GPAction)

Number of refactoring operations in the solution = 6 **Number of refactoring operations in the expected model** = 6 **Number of common refactoring operations (similarity)** = 6 **Precision** = 1.0 **Recall** = 1.0

Solution in the 9th execution	
Block 1	**CIM**: net.sourceforge.ganttproject.ResourceTreeTable
	CBE: com.mxgraph.view.mxLayoutManager
	The obtained SOR: Pull_up_method(createPopup();net.sourceforge.ganttproject.ResourceTreeTable; GPTreeTableBase) Pull_up_field(popupMenu;net.sourceforge.ganttproject.ResourceTreeTable; GPTreeTableBase)
	The expected SOR: Pull_up_method(isExpanded; ResourceTreeTable; GPTreeTableBase) Pull_up_field(clickPoint; ResourceTreeTable; GPTreeTableBase)
Block 2	**CIM**: net.sourceforge.ganttproject.GanttProject
	CBE: com.mxgraph.swing.handler.mxGraphHandler
	The obtained SOR: Pull_up_method(getToolBar; net.sourceforge.ganttproject.GanttProject; GanttProjectBase) Pull_up_field(myDocumentManager;net.sourceforge.ganttproject.GanttProject; GanttProjectBase)
	The expected SOR: Pull_up_method(getViewIndex; GanttProject; GanttProjectBase) Pull_up_field(myRolloverActions; GanttProject; GanttProjectBase)
Block 3	**CIM**: net.sourceforge.ganttproject.action.NewArtefactAction
	CBE: com.mxgraph.layout.mxStackLayout
	The obtained SOR: Pull_up_field(myIconOnMouseOver;net.sourceforge.ganttproject.action.NewArtefactAction; GPAction) Pull_up_field(myProvider;net.sourceforge.ganttproject.action.NewArtefactAction; GPAction)
	The expected SOR: Pull_up_field(myIconOnMouseOver; NewArtefactAction; GPAction) Pull_up_field(myIconVisible; NewArtefactAction; GPAction)
Block 4	**CIM**: net.sourceforge.ganttproject.GanttGraphicArea
	CBE: com.mxgraph.util.mxLine
	The obtained SOR: Pull_up_method(getName();net.sourceforge.ganttproject.GanttGraphicArea; ChartComponentBase)
	The expected SOR: Pull_up_method(getProject; GanttGraphicArea;ChartComponentBase)

	Pull_up_field(myTaskImageGenerator; GanttGraphicArea; ChartComponentBase)
Block 5	**CIM**: net.sourceforge.ganttproject.chart.ResourceLoadRenderer
	CBE: com.mxgraph.shape.mxImageShape
	The obtained SOR: Pull_up_method(getChartEndDate();net.sourceforge.ganttproject.chart.ResourceLoadRenderer;ChartRendererBase) Pull_up_field(myDistributions;net.sourceforge.ganttproject.chart.ResourceLoadRenderer; ChartRendererBase) Pull_up_method(beforeProcessingTimeFrames();net.sourceforge.ganttproject.chart.ResourceLoadRenderer; ChartRendererBase)
	The expected SOR: Pull_up_method(getChartStartDate; ResourceLoadRenderer; ChartRendererBase) Pull_up_field(myResourcechart; ResourceLoadRenderer; ChartRendererBase)
Block 6	**CIM**: net.sourceforge.ganttproject.gui.options.model.DefaultDateOption
	CBE: java.net.sf.jabref.collab.EntryChange
	The obtained SOR: Pull_up_method(loadPersistentValue();net.sourceforge.ganttproject.gui.options.model.DefaultDateOption; GPAbstractOption)
	The expected SOR: Pull_up_method(loadPersistentValue; DefaultDateOption; GPAbstractOption)
Number of refactoring operations in the solution = 11 **Number of refactoring operations in the expected model** = 11 **Number of common refactoring operations (similarity)** = 10 **Precision** = 0.9090909090909091 **Recall** = 0.9090909090909091	

Solution in the 18th execution	
Block 1	**CIM**: net.sourceforge.ganttproject.io.VacationSaver
	CBE: java.net.sf.jabref.collab.EntryChange
	The obtained SOR: Pull_up_method(save(); net.sourceforge.ganttproject.io.VacationSaver; SaverBase)
	The expected SOR: Pull_up_method(save(); VacationSaver; SaverBase)
Block 2	**CIM**: net.sourceforge.ganttproject.action.NewArtefactAction
	CBE: com.mxgraph.view.mxCellState
	The obtained SOR: Pull_up_method(actionPerformed();net.sourceforge.ganttproject.action.NewArtefactAction; GPAction)

	The expected SOR: Pull_up_field(myIconOnMouseOver; NewArtefactAction; GPAction) Pull_up_method(isIconVisible; NewArtefactAction; GPAction)
Block 3	**CIM**: net.sourceforge.ganttproject.action.RefreshViewAction
	CBE: java.net.sf.jabref.export.layout.format.WrapFileLinks
	The obtained SOR: Pull_up_field(myUIFacade;net.sourceforge.ganttproject.action.RefreshViewAction; GPAction)
	The expected SOR: Pull_up_field(myUIFacade; RefreshViewAction; GPAction)
Number of refactoring operations in the solution = 3 **Number of refactoring operations in the expected model** = 4 **Number of common refactoring operations (similarity)** = 3 **Precision** = 1.0 **Recall** = 0.75	

Solution in the 27th execution	
Block 1	**CIM**: net.sourceforge.ganttproject.GanttCalendar
	CBE: com.mxgraph.swing.handler.mxInsertHandler
	The obtained SOR: Pull_up_field(language; net.sourceforge.ganttproject.GanttCalendar; GregorianCalendar) Pull_up_field(isFixed; net.sourceforge.ganttproject.GanttCalendar; GregorianCalendar) Pull_up_method(getWeek();net.sourceforge.ganttproject.GanttCalendar; GregorianCalendar)
	The expected SOR: Pull_up_field(isXMLString; GanttCalendar; GregorianCalenda1r) Pull_up_field(isFixed; GanttCalendar;GregorianCalenda1r) Pull_up_method(module; GanttCalendar; GregorianCalenda1r)
Block 2	**CIM**: net.sourceforge.ganttproject.importer.FileChooserPage
	CBE: java.net.sf.jabref.groups.AllEntriesGroup
	The obtained SOR: Pull_up_method(getTitle();net.sourceforge.ganttproject.importer.FileChooserPage; FileChooserPageBase)
	The expected SOR: Pull_up_method(getTitle; FileChooserPage; FileChooserPageBase) Pull_up_field(myWebPublishingGroup; FileChooserPage; FileChooserPageBase)
Block 3	**CIM**: net.sourceforge.ganttproject.action.RedoAction
	CBE: com.mxgraph.util.svg.PathParser
	The obtained SOR: Pull_up_method(isIconVisible();net.sourceforge.ganttproject.action.RedoAction; GPAction)

	Pull_up_method(getLocalizedName();net.sourceforge.ganttproject.action.RedoAction; GPAction)
	The expected SOR: Pull_up_method(isIconVisible; RedoAction; GPAction)
Block 4	**CIM**:net.sourceforge.ganttproject.gui.options.model.DefaultEnumerationOption
	CBE: java.net.sf.jabref.journals.ManageJournalsAction
	The obtained SOR: Pull_up_field(myValue;net.sourceforge.ganttproject.gui.options.model.DefaultEnumerationOption; GPAbstractOption)
	The expected SOR: Pull_up_field(myValue; DefaultEnumerationOption; GPAbstractOption)
Block 5	**CIM**: net.sourceforge.ganttproject.action.RefreshViewAction
	CBE: java.net.sf.jabref.collab.PreambleChange
	The obtained SOR: Pull_up_method(RefreshViewAction();net.sourceforge.ganttproject.action.RefreshViewAction; GPAction) Pull_up_field(myUIFacade;net.sourceforge.ganttproject.action.RefreshViewAction; GPAction)
	The expected SOR: Pull_up_field(myUIFacade; RefreshViewAction; GPAction)
Block 6	**CIM**: net.sourceforge.ganttproject.document.HttpDocument
	CBE: com.mxgraph.layout.mxPartitionLayout
	The obtained SOR: Pull_up_field(webdavResource;net.sourceforge.ganttproject.document.HttpDocument; AbstractURLDocument) Pull_up_method(getInputStream();net.sourceforge.ganttproject.document.HttpDocument; AbstractURLDocument)
	The expected SOR: Pull_up_field(myValue; DefaultEnumerationOption; GPAbstractOption)
Number of refactoring operations in the solution = 11 **Number of refactoring operations in the expected model** = 9 **Number of common refactoring operations (similarity)** = 8 **Precision** = 0.7272727272727273 **Recall** = 0.8888888888888888	

ANNEX III

PARETO FRONT SOLUTIONS IN XERCES 2.7

Solution 1	**Semantic similarity = 0.35** **Structural similarity = 0.8**
Block 1	**CIM**: org.apache.xerces.dom.DOMLocatorImpl
	CBE: CH.ifa.draw.standard.FigureTransferCommand
	The obtained SOR: Add_parameter(newParameter; getLineNumber[ ]) Remove_parameter(columnNumber; DOMLocatorImpl())
Block 2	**CIM**: org.apache.xerces.dom.RangeImpl
	CBE: CH.ifa.draw.standard.ToggleGridCommand
	The obtained SOR: Add_parameter(newParameter; isAncestorOf[a b])
Block 3	**CIM**: org.apache.xerces.util.DOMEntityResolverWrapper
	CBE: CH.ifa.draw.contrib.TriangleRotationHandle
	The obtained SOR: Remove_parameter(entityResolver;DOMEntityResolverWrapper()) Add_parameter(newParameter;DOMEntityResolverWrapper[entityResolver])
Block 4	**CIM**: org.apache.xml.serialize.XMLSerializer
	CBE: CH.ifa.draw.figures.TextFigure
	The obtained SOR: Add_parameter(newParameter; printText[text preserveSpace unescaped])
Block 5	**CIM**: org.apache.xerces.impl.xs.util.StringListImpl
	CBE: CH.ifa.draw.standard.ChangeAttributeCommand
	The obtained SOR: Rename_method(StringListImpl(); newStringListImpl())
Block 6	**CIM**: org.apache.xerces.impl.xs.util.NSItemListImpl
	CBE : CH.ifa.draw.figures.ScribbleTool
	The obtained SOR: Add_parameter(newParameter; item[index])

Solution 2	**Semantic similarity = 0.25** **Structural similarity = 0.85**
Block 1	**CIM**: org.apache.xerces.impl.dv.xs.AbstractDateTimeDV
	CBE: CH.ifa.draw.contrib.PolygonTool
	The obtained SOR: Remove_parameter(finalValue; cloneDate())
Block 2	**CIM**: org.apache.xerces.dom.DOMLocatorImp
	CBE: CH.ifa.draw.standard.ChangeAttributeCommand
	The obtained SOR:

	Rename_method(DOMLocatorImpl(); newDOMLocatorImpl())
Block 3	**CIM**: org.apache.xerces.impl.XMLNSDocumentScannerImpl
	CBE: CH.ifa.draw.contrib.TriangleRotationHandle
	The obtained SOR: Remove_parameter(componentManager; reset()) Add_parameter(newParameter; scanEndElement[ ])

Solution 3	**Semantic similarity = 0.45** **Structural similarity = 0.77**
Block 1	**CIM**: org.apache.xerces.impl.xpath.regex.RegularExpression
	CBE: CH.ifa.draw.figures.InsertImageCommand
	The obtained SOR: Remove_parameter(pattern; equals())
Block 2	**CIM**: org.apache.xerces.dom.CoreDOMImplementationImpl
	CBE: CH.ifa.draw.applet.DrawApplet
	The obtained SOR: Rename_method(createDOMInput(); newcreateDOMInput()) Extract_interface(org.apache.xerces.dom.CoreDOMImplementationImpl; ICoreDOMImplementationImpl)
Block 3	**CIM**: org.apache.xerces.impl.xs.util.StringListImpl
	CBE: CH.ifa.draw.contrib.TriangleFigure
	The obtained SOR: Rename_method(getLength(); newgetLength())
Block 4	**CIM**: org.apache.xerces.impl.XMLNSDocumentScannerImpl
	CBE: CH.ifa.draw.contrib.TriangleFigure
	The obtained SOR: Rename_method(checkDuplicates(); newcheckDuplicates())

Solution 4	**Semantic similarity = 0.51** **Structural similarity = 0.7**
Block 1	**CIM**: org.apache.xerces.impl.dv.xs.MonthDayDV
	CBE: CH.ifa.draw.figures.TextFigure
	The obtained SOR: Add_parameter(newParameter; parse[str])
Block 2	**CIM**: org.apache.xerces.impl.xs.traversers.XSDElementTraverser
	CBE: CH.ifa.draw.samples.javadraw.MySelectionTool
	The obtained SOR: Add_parameter(newParameter; traverseLocal[elmDecl schemaDoc grammar allContextFlags enclosingCT])
Block 3	**CIM**: org.apache.xerces.impl.xs.identity.XPathMatcher
	CBE: CH.ifa.draw.figures.ConnectedTextTool
	The obtained SOR: Add_parameter(newParameter; handleContent[eDecl value])
Block 4	**CIM**: org.apache.xerces.impl.dv.xs.XSSimpleTypeDecl

	CBE: CH.ifa.draw.standard.ChangeAttributeCommand
	The obtained SOR: Rename_method(getFixedFacets(); newgetFixedFacets())
Block 5	**CIM**: org.apache.xerces.dom.CoreDocumentImpl
	CBE: CH.ifa.draw.contrib.SplitPaneDrawApplication
	The obtained SOR: Remove_parameter(set;setMutationEvents()) Add_parameter(newParameter; getElementsByTagName[tagname])
Block 6	**CIM**: org.apache.xerces.impl.dv.xs.DayDV
	CBE: CH.ifa.draw.samples.javadraw.MySelectionTool
	The obtained SOR: Add_parameter(newParameter; dateToString[date])

Solution 5	**Semantic similarity = 0.58** **Structural similarity = 0.62**
Block 1	**CIM**: org.apache.xerces.impl.dv.xs.ListDV
	CBE: CH.ifa.draw.standard.StandardDrawingView
	The obtained SOR: We can not apply Move_Method refactoring Rename_method(length(); newlength()) Replace_inheritance_with_delegation(ListDV; TypeValidator; Delegation) Rename_method(toString(); newtoString())
Block 2	**CIM**: org.apache.xerces.impl.dv.xs.YearMonthDV
	CBE: CH.ifa.draw.standard.ActionTool
	The obtained SOR: Add_parameter(newParameter; dateToString[date]) Remove_parameter(date; dateToString())

Solution 6	**Semantic similarity = 0.64** **Structural similarity = 0.56**
Block 1	**CIM**: org.apache.xerces.impl.xs.identity.XPathMatcher
	CBE: CH.ifa.draw.applet.DrawApplet
	The obtained SOR: Rename_method(startElement(); newstartElement()) Extract_interface(org.apache.xerces.impl.xs.identity.XPathMatcher; IXPathMatcher)
Block 2	**CIM**: org.apache.xerces.impl.xs.ElementPSVImpl
	CBE: CH.ifa.draw.standard.SendToBackCommand
	The obtained SOR: Add_parameter(newParameter; getElementDeclaration[ ])
Block 3	**CIM**: org.apache.xerces.impl.xpath.XPathException
	CBE: CH.ifa.draw.standard.StandardDrawingView

	The obtained SOR: We can not apply Move_Method refactoring Rename_method(XPathException(); newXPathException()) Replace_inheritance_with_delegation(XPathException;Exception; Delegation) Rename_method(getKey(); newgetKey())

Solution 7	**Semantic similarity = 0.7** **Structural similarity = 0.5**
Block 1	**CIM**: org.apache.xerces.impl.dv.xs.HexBinaryDV
	CBE: CH.ifa.draw.figures.UngroupCommand
	The obtained SOR: Remove_parameter(obj; equals()) Rename_method(getAllowedFacets(); newgetAllowedFacets())
Block 2	**CIM**: org.apache.xerces.xinclude.XIncludeHandler
	CBE: CH.ifa.draw.standard.CopyCommand
	The obtained SOR: Rename_method(endAttlist(); newendAttlist())
Block 3	**CIM**: org.apache.xerces.impl.xs.opti.DefaultText
	CBE: CH.ifa.draw.standard.CreationTool
	The obtained SOR: Add_parameter(newParameter; substringData[offset count]) Rename_method(substringData(); newsubstringData())
Block 4	**CIM**: org.apache.xerces.impl.XMLDocumentScannerImpl
	CBE: CH.ifa.draw.standard.ConnectionTool
	The obtained SOR: Remove_parameter(inputSource; setInputSource()) Rename_method(dispatch(); newdispatch())

Solution 8	**Semantic similarity = 0.8** **Structural similarity = 0.22**
Block 1	**CIM**: org.apache.xerces.xinclude.XIncludeHandler
	CBE: CH.ifa.draw.contrib.SplitPaneDrawApplication
	The obtained SOR: Remove_parameter(augs; endDocument()) Add_parameter(newParameter; endParameterEntity[name augmentations])
Block 2	**CIM**: org.apache.xerces.impl.xs.opti.SchemaDOM
	CBE: CH.ifa.draw.figures.InsertImageCommand
	The obtained SOR: Remove_parameter(node; traverse())
Block 3	**CIM**: org.apache.wml.dom.WMLDOMImplementationImpl
	CBE: CH.ifa.draw.standard.ChangeAttributeCommand
	The obtained SOR: Rename_method(createDocument(); newcreateDocument())

Block 4	**CIM**: org.apache.xerces.impl.XMLEntityManager
	CBE: CH.ifa.draw.contrib.TriangleRotationHandle
	The obtained SOR: Remove_parameter(baseSystemId; addExternalEntity()) Add_parameter(newParameter; print[currentEntity])

Solution 9	**Semantic similarity = 0.58** **Structural similarity = 0.62**
Block 1	**CIM**: org.apache.xerces.impl.xs.opti.SchemaDOM
	CBE: CH.ifa.draw.standard.ChangeAttributeCommand
	The obtained SOR: Rename_method(startAnnotationCDATA(); newstartAnnotationCDATA())
Block 2	**CIM**: org.apache.xerces.impl.xs.SubstitutionGroupHandler
	CBE: CH.ifa.draw.standard.SelectionTool
	The obtained SOR: Remove_parameter(methods; getSubGroupB()) Add_parameter(newParameter; addSubstitutionGroup[elements])
Block 3	**CIM**: org.apache.xerces.impl.xs.XMLSchemaLoader
	CBE: CH.ifa.draw.figures.LineConnection
	The obtained SOR: Remove_parameter(entityResolver; XMLSchemaLoader())
Block 4	**CIM**: org.apache.xerces.impl.xs.traversers.XSDHandler
	CBE: CH.ifa.draw.standard.ConnectionTool
	The obtained SOR: Remove_parameter(decl; findXSDocumentForDecl()) Rename_method(prepareForTraverse(); newprepareForTraverse())
Block 5	**CIM**: org.apache.xerces.dom.PSVIDOMImplementationImpl
	CBE: CH.ifa.draw.standard.HandleTracker
	The obtained SOR: Add_parameter(newParameter; createDocument[namespaceURI qualifiedName doctype]) Remove_parameter(qualifiedName; createDocument())

Solution 10	**Semantic similarity = 0.77** **Structural similarity = 0.35**
Block 1	**CIM**: org.apache.xerces.dom.DOMLocatorImpl
	CBE: CH.ifa.draw.contrib.PolygonScaleHandle
	The obtained SOR: Add_parameter(newParameter; DOMLocatorImpl[lineNumber columnNumber offset relatedData uri]) Remove_parameter(columnNumber; DOMLocatorImpl())
Block 2	**CIM**: org.apache.xml.serialize.XML11Serializer
	CBE: CH.ifa.draw.samples.javadraw.MySelectionTool
	The obtained SOR:

	Add_parameter(newParameter; printCDATAText[text])
Block 3	**CIM**: org.apache.xerces.impl.xs.SubstitutionGroupHandler
	CBE: CH.ifa.draw.standard.FigureTransferCommand
	The obtained SOR: Add_parameter(newParameter; reset[ ]) Remove_parameter(element; inSubstitutionGroup())

BIBLIOGRAPHY

Akroyd, M. 1996. *AntiPatterns Session Notes*. San Francisco: Object World West.

Alikacem, E. H., and H. Sahraoui. 2006. « Détection d'anomalies utilisant un langage de description de règle de qualité ». In *Actes du 12ème colloque LMO*.

Arcuri, A., and L. Briand. 2012. « A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering ». *Software Testing, Verification and Reliability,* vol. 24, p. 219-250.

Basili, V., S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård and M. Zelkowitz. 1996. « The empirical investigation of Perspective-Based Reading ». *Empirical Software Engineering,* vol. 1, n^{o} 2, p. 133-164.

Bavota, G., F. Carnevale, A. D. Lucia, M. D. Penta and R. Oliveto. 2012. « Putting the developer in-the-loop: an interactive GA for software re-modularization ». In *Proceedings of the 4th international conference on Search Based Software Engineering*. (Riva del Garda, Italy, 28-30 September), p. 75-89. Springer-Verlag.

Bellur, U., and V. Vallieswaran. 2006. « On OO Design Consistency in Iterative Development ». In *Proceedings of the Third International Conference on Information Technology: New Generations (ITNG)*. (Las Vegas, Nevada, USA, 10-12 April), p. 46-51. IEEE Computer Society.

Ben Fadhel, A., M. Kessentini, P. Langer and M. Wimmer. 2012. « Search-based detection of high-level model changes ». In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*. (Riva del Garda, Trento, Italy, 23-30 September), p. 212-221. IEEE Computer Society.

Berenbach, B. 2004. « The evaluation of large, complex UML analysis and design models ». In *Proceedings of 26th International Conference on Software Engineering (ICSE)*. (Edinburgh, UK, 23-28 May), p. 232-241. IEEE Computer Society.

Biermann, E. 2010. « EMF model transformation based on graph transformation: formal foundation and tool environment ». In *Proceedings of the 5th International Conference on Graph Transformations (ICGT)*. (Enschede, The Netherlands, 27 September-2 October), p. 381-383. Springer-Verlag.

Brown, J. W., C. M. Raphael, W. S. M. Hays and J. M. Thomas. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, WILEY. 336 p.

Bull, R. I. 2008. « Model driven visualization: towards a model driven engineering approach for information visualization ». Thesis in Software Engineering. University of Victoria, 248 p.

Chen, Y.-p. 2007. « Interactive music composition with the CFE framework ». *SIGEVOlution,* vol. 2, n° 1, p. 9-16.

Davis, R., B. Buchanan and E. Shortliffe. 1977. « Production rules as a representation for a knowledge-based consultation program ». *Artificial Intelligence,* vol. 8, n° 1, p. 15-45.

Dawkins, R. 1986. *The BlindWatchmaker*, 1st edition. Longman, Essex, U.K., 358 p.

Deb, K., A. Pratap, S. Agarwal and T. Meyarivan. 2002. « A fast and elitist multiobjective genetic algorithm: NSGA-II ». *IEEE Transactions on Evolutionary Computation,* vol. 6, n° 2, p. 182-197.

Dhambri, K., H. Sahraoui and P. Poulin. 2008. « Visual Detection of Design Anomalies ». In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*. (Athens, Greece, 1-4 April), p. 279-283.

Douglas, C. S. 2006. « Guest Editor's Introduction: Model-Driven Engineering ». *IEEE Computer,* vol. 39, n° 2, p. 41-47.

Du Bois, B., S. Demeyer and J. Verelst. 2004. « Refactoring " Improving Coupling and Cohesion of Existing Code ». In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)* (Delft University of Technology, Netherlands, 9-12 November), p. 144-151. IEEE Computer Society.

El-Boussaidi, G., and H. Mili. 2008. « Detecting Patterns of Poor Design Solutions Using Constraint Propagation ». In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. (Toulouse, France, 28 September-3 October), p. 189-203. Springer-Verlag.

El-Boussaidi, G., and H. Mili. 2011. « Understanding design patterns — what is the problem? ». *Software: Practice and Experience,* vol. 42, p. 1495-1529.

Erni, K., and C. Lewerentz. 1996. « Applying design-metrics to object-oriented frameworks ». In *Proceedings of the 3rd International Software Metrics Symposium*. (Washington, DC, USA, 25-26 March), p. 64-74. IEEE Computer Society.

Fenton, N., and S. L. Pfleeger. 1998. *Software Metrics: A Rigorous and Practical Approach*, 2nd Edition. Boston, MA, USA: PWS Publishing Co., 656 p.

Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 455 p.

Fowler, M., and K. Beck. 1999. « Refactoring: Improving the Design of Existing Code ». In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods* (Chicago, USA, 4-7 August), p. 256. Springer-Verlag.

Gamma, E., R. Helm, J. Ralph and J. Vlissides. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 383 p.

Genero, M., M. Piattini and C. Calero. 2002. « Empirical validation of class diagram metrics ». In *Proceedings of the International Symposium in Empirical Software Engineering (ISESE)*. (Nara, Japan, 3-4 October), p. 195-203.

Ghannem, A., G. El-Boussaidi and M. Kessentini. 2013. « Model Refactoring Using Interactive Genetic Algorithm ». In *Proceedings of the 5th Symposium on Search Based Software Engineering (SSBSE)*. (24-26 August), sous la dir. de Ruhe, Günther, and Yuanyuan Zhang Vol. 8084, p. 96-110. Coll. « Lecture Notes in Computer Science »: Springer Berlin Heidelberg.

Ghannem, A., G. El-Boussaidi and M. Kessentini. 2014a. « A Design Defect Example Is Worth a Dozen Detection Rules ». *Software Quality Journal (Accepted)*.

Ghannem, A., G. El-Boussaidi and M. Kessentini. 2014b. « Example-based Model Refactoring using Multi-Objective Optimization ». *Journal of Automated Software Engineering (Submitted)*.

Ghannem, A., G. El-Boussaidi and M. Kessentini. 2014c. « Model refactoring using examples: a search-based approach ». *Journal of Software: Evolution and Process,* vol. 26, p. 692-713.

Ghannem, A., M. Kessentini and G. El-Boussaidi. 2011. « Detecting Model Refactoring Opportunities Using Heuristic Search ». In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*. (IBM Corp., Riverton, NJ, USA), p. 175-187. Marin Litoiu, Eleni Stroulia, and Stephen MacKay (Eds.).

Gheyi, R., T. Massoni and P. Borba. 2007. « A Static Semantics for Alloy and its Impact in Refactorings ». *Electronic Notes in Theoretical Computer Science,* vol. 184, n^{o} 0, p. 209-233.

Goldberg, E. D. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc, 372 p.

Gueheneuc, Y.-G., and A. Amiot. 2004. « Recovering binary class relationships: putting icing on the UML cake ». In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*. (Vancouver, BC, Canada, 24-28 October). ACM.

Gueheneuc, Y.-G., H. Sahraoui and F. Zaidi. 2004. « Fingerprinting design patterns ». In *Proceedings of 11th Working Conference onReverse Engineering (WCRE)*. (Delft, The Netherlands, 8-12 November), p. 172-181. IEEE Computer Society.

Harman, M., and J. Clark. 2004. « Metrics are fitness functions too ». In *Proceedings of the 10th International Symposium on Software Metrics*. (Chicago, IL, USA, 11-17 September), p. 58-69. IEEE Computer Society.

Harman, M., and L. Tratt. 2007. « Pareto optimal search based refactoring at the design level ». In *Proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO)*. (London, England, UK, 07-11 July), p. 1106-1113. ACM.

Heckel, R. 1995. « Algebraic graph transformations with application conditions ». TU Berlin.

Hoel, G. P. 1954. *Introduction to Mathematical Statistics*. Wiley.

Howe, C. D. 2009. « RiTa: creativity support for computational literature ». In *Proceedings of the 7th ACM conference on Creativity and cognition*. (Berkeley, CA, USA, 27-30 October), p. 205-210. ACM.

ISO/IEC. 2006. « International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering; Software Life Cycle Processes &; Maintenance ». *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, p. 01-46.

Jensen, C. A., and H. C. B. Cheng. 2010. « On the use of genetic programming for automated refactoring and the introduction of design patterns ». In *Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO)*. (Portland, OR, USA, July 07-11 July), p. 1341-1348. 1830731: ACM.

Kataoka, Y., D. Notkin, D. M. Ernst and G. W. Griswold. 2001. « Automated support for program refactoring using invariants ». In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)* (Florence, ITALY, 6-10 November), p. 736-743. IEEE Computer Society.

Kerievsky, J. 2004. *Refactoring to patterns*. Addison-Wesley, 336 p.

Kessentini, M., W. Kessentini, H. Sahraoui, M. Boukadoum and A. Ouni. 2011a. « Design Defects Detection and Correction by Example ». In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*. (Kingston, ON, CANADA, 22-24 June), p. 81-90. IEEE Computer Society.

Kessentini, M., H. Sahraoui and M. Boukadoum. 2008. « Model Transformation as an Optimization Problem ». In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*. (Toulouse, France, 28 September-3 October), p. 159-173. Springer-Verlag.

Kessentini, M., H. Sahraoui, M. Boukadoum and O. Omar. 2012. « Search-based model transformation by example ». *Software & Systems Modeling,* vol. 11, n° 2, p. 209-226.

Kessentini, M., H. Sahraoui, M. Boukadoum and M. Wimmer. 2011b. « Search-based design defects detection by example ». In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software* (Saarbrücken, Germany, 26 March-3 April), p. 401-415. Springer-Verlag.

Kessentini, M., S. Vaucher and H. Sahraoui. 2010. « Deviance from perfection is a better criterion than closeness to evil when identifying risky code ». In *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE)*. (Antwerp, Belgium, 20-24 September), p. 113-122. ACM.

Kessentini, W., M. Kessentini, H. Sahraoui, S. Bechikh and A. Ouni. 2014. « A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection ». *Software Engineering, IEEE Transactions on,* vol. 40, n° 9, p. 841-861.

Khomh, F., S. Vaucher, Y.-G. Gueheneuc and H. Sahraoui. 2009. « A Bayesian Approach for the Detection of Code and Design Smells ». In *Proceedings of the 9th International Conference on Quality Software (QSIC)* (Jeju, Korea, 24-25 August), p. 305-314. IEEE Computer Society.

Kim, H.-S., and S.-B. Cho. 2000. « Application of interactive genetic algorithm to fashion design ». *Engineering Applications of Artificial Intelligence,* vol. 13, n° 6, p. 635-644.

Kim, M., M. Gee, A. Loh and N. Rachatasumrit. 2010. « Ref-Finder: a refactoring reconstruction tool based on logic query templates ». In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*. (Santa Fe, New Mexico, USA, 7-11 November), p. 371-372. ACM.

Kirkpatrick, S., C. D. Gelatt and M. P. Vecchi. 1983. « Optimization by simulated annealing ». *Science,* vol. 220, n° 4598, p. 671-680.

Kothari, S. C., L. Bishop, J. Sauceda and G. Daugherty. 2004. « A Pattern-Based Framework for Software Anomaly Detection ». *Software Quality Journal,* vol. 12, p. 99-120.

Koza, R. J. 1992. *Genetic programming: on the programming of computers by means of natural selection* Cambridge, MA, USA: MIT Press, 680 p.

Laitenberger, O., C. Atkinson, M. Schlich and K. El Emam. 2000. « An experimental comparison of reading techniques for defect detection in UML design documents ». *J. Syst. Softw.,* vol. 53, n° 2, p. 183-204.

Lange, C. F. J., and M. R. V. Chaudron. 2006. « Effects of defects in UML models: an experimental investigation ». In *Proceedings of the 28th international conference on Software engineering*. (Shanghai, China, 20-28 May), p. 401-411. ACM.

Langelier, G., H. Sahraoui and P. Poulin. 2005. « Visualization-based analysis of quality for large-scale software systems ». In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE)*. (Long Beach, CA, USA, 7-11 November), p. 214-223. ACM.

Leung, F., and N. Bolloju. 2005. « Analyzing the Quality of Domain Models Developed by Novice Systems Analysts ». In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS)* (Big Island, HI, USA, 03-06 January), p. 188b. IEEE Computer Society.

Lientz, B. P., E. B. Swanson and G. E. Tompkins. 1978. « Characteristics of application software maintenance ». *Commun. ACM,* vol. 21, n° 6, p. 466-471.

Lindland, O. I., G. Sindre and A. Solvberg. 1994. « Understanding quality in conceptual modeling ». *Software, IEEE,* vol. 11, n° 2, p. 42-49.

Liu, H., L. Yang, Z. Niu, Z. Ma and W. Shao. 2009. « Facilitating software refactoring with appropriate resolution order of bad smells ». In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. (Amsterdam, The Netherlands, 24-28 August), p. 265-268. ACM.

Mantyla, M., V. Jari and L. Casper. 2003. « A Taxonomy and an Initial Empirical Study of Bad Smells in Code ». In *Proceedings of the International Conference on Software Maintenance (ICSM)*. (Amsterdam, The Netherlands, 22-26 September), p. 381. 943571: IEEE Computer Society.

Marinescu, R. 2004. « Detection strategies: metrics-based rules for detecting design flaws ». In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*. (Chicago, IL, USA, 11-17 September), p. 350-359. IEEE Computer Society.

Mens, T., G. Taentzer and M. Dirk. 2007a. « Challenges in Model Refactoring ». In *Proceedings of the 1st Workshop on Refactoring Tools (WRT)*. (University of Berlin, Germany, 31 July).

Mens, T., G. Taentzer and O. Runge. 2007b. « Analysing refactoring dependencies using graph transformation ». *Software and Systems Modeling,* vol. 6, n° 3, p. 269-285.

Mens, T., D. Tamzalit, M. Hoste and J. Pinna Puissant. 2010. « Amélioration de la qualité de modèles: Une étude de deux approches complémentaires ». *Revue Technique et Science Informatiques (TSI)- Numéro Spécial IDM*.

Mens, T., and T. Tourwé. 2004. « A Survey of Software Refactoring ». *IEEE Trans. Softw. Eng.,* vol. 30, n° 2, p. 126-139.

Miceli, T., H. Sahraoui and R. Godin. 1999. « A Metric Based Technique for Design Flaws Detection and Correction ». In *Proceedings of the 14th IEEE international conference on Automated Software Engineering (ASE)*. (Cocoa Beach, Florida, USA, 12-15 October), p. 307. IEEE Computer Society.

Mitchell, M. 1998. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 209 p.

Moha, N. 2008. « DECOR : Détection et correction des défauts dans les systèmes orientés objet ». Montréal, Université de Montréal & Université des Sciences et Technologies de Lille 157 p.

Moha, N., Y.-G. Gueheneuc, L. Duchien and A. F. Le Meur. 2010. « DECOR: A Method for the Specification and Detection of Code and Design Smells ». *IEEE Transactions Software Engineering,* vol. 36, n° 1, p. 20-36.

Moha, N., Y.-G. Gueheneuc and P. Leduc. 2006. « Automatic Generation of Detection Algorithms for Design Defects ». In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE)* (18-22 Sept. 2006), p. 297-300. IEEE/ACM

Moha, N., V. Mahé, O. Barais and J.-M. Jézéquel. 2009. « Generic Model Refactorings ». In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*. (Denver, CO, USA, 4-9 October), p. 628-643. Springer-Verlag.

Moha, N., A. M. Rouane Hacene, Petko V. and Y. G. Gueheneuc. 2008a. « Refactorings of design defects using relational concept analysis ». In *Proceedings of the 6th international conference on Formal concept analysis*. (Montreal, Canada), p. 289-304. 1787767: Springer-Verlag.

Moha, N., A. M. Rouane Hacene, P. Valtchev and Y.-G. Guéhéneuc. 2008b. « Refactorings of Design Defects Using Relational Concept Analysis ». In, sous la dir. de Medina, Raoul, and Sergei Obiedkov. Vol. 4933, p. 289-304. Springer Berlin Heidelberg.

Munro, M. J. 2005. « Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code ». In *Proceddings of the 11th International Symposium on Software Metrics* (Como, Italy, 19-22 September), p. 15-15. IEEE.

O' Cinnéide, M., L. Tratt, M. Harman, S. Counsell and I. H. Moghadam. 2012. « Experimental assessment of software metrics using automated refactoring ». In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. (Lund, Sweden, 19-20 September), p. 49-58. ACM.

O'Keeffe, M. 2008. « Search-based refactoring: an empirical study ». *Journal of Software : Maintenance and Evolution (JSME),* vol. 20, n° 5, p. 345-364.

O'Keeffe, M., and M. O'Cinneide. 2006. « Search-based software maintenance ». In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR)* (Bari, Italy, 22-24 March), p. 10 pp.-260.

O'Keeffe., M., and M. O. Cinneide. 2008. « Search-based refactoring for software maintenance ». *Journal of System and Software,* vol. 81, n° 4, p. 502-516.

Opdyke, F. W. 1992. « Refactoring : A Program Restructuring Aid in Designing Object-Oriented Application Frameworks ». University of Illinois at Urbana-Champaign.

Ouni, A., M. Kessentini, H. Sahraoui and M. Boukadoum. 2013. « Maintainability defects detection and correction: a multi-objective approach ». *Journal of Automated Software Engineering,* vol. 20, n° 1, p. 47-79.

Paradigm, V. 2008. « http://www.visual-paradigm.com/product/vpuml/ ».

Pearl, J. 1984. *Heuristics: intelligent search strategies for computer problem solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 382 p.

Pressman, S. R. 2001. *Software Engineering - A Practitioner's Approach*, 5th Edition. McGraw-Hill Higher Education.

Pretschner, A., and W. Prenninger. 2007. « Computing refactorings of state machines ». *Software and Systems Modeling,* vol. 6, n° 4, p. 381-399.

Qayum, F., and R. Heckel. 2009. « Local Search-Based Refactoring as Graph Transformation ». In *Proceddings of the 1st International Symposium on Search Based Software Engineering (SSBSE)*. (Cumberland Lodge, Windsor, UK, 13-15 May), p. 43-46. Springer Berlin Heidelberg.

Ragnhild, V. D. S., and M. D'Hondt. 2006. « Model refactorings through rule-based inconsistency resolution ». In *Proceedings of the 2006 ACM symposium on Applied computing*. (Dijon, France, 23-27 April), p. 1210-1217. 1141564: ACM.

Ragnhild, V. D. S., V. Jonckers and T. Mens. 2007. « A formal approach to model refactoring and model refinement ». *Software and Systems Modeling (SoSyM),* vol. 6, n° 2, p. 139-162.

Reimann, J., M. Seifert and U. Aßmann. 2010. « Role-Based Generic Model Refactoring ». In *Model Driven Engineering Languages and Systems*. (Oslo, Norway, 3-8 October), sous la dir. de Petriu, DorinaC, Nicolas Rouquette and Øystein Haugen Vol. 6395, p. 78-92. Coll. « Lecture Notes in Computer Science »: Springer Berlin Heidelberg.

Riel, J. A. 1996. *Object-Oriented Design Heuristics*. Addison Wesley.

Saaty, T. L. 1985. « Decision making for leaders ». *Systems, Man and Cybernetics, IEEE Transactions on,* vol. SMC-15, n° 3, p. 450-452.

Sahin, D., M. Kessentini, S. Bechikh and K. Deb. 2014. « Code-Smell Detection as a Bilevel Problem ». *ACM Trans. Softw. Eng. Methodol.,* vol. 24, n° 1, p. 1-44.

Seacord, C. R., V. D. Plakosh and A. G. Lewis. 2003. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Boston, MA, USA Addison-Wesley Longman Publishing Co., Inc., 368 p.

Seng, O., J. Stammel and D. Burkhart. 2006. « Search-based determination of refactorings for improving the class structure of object-oriented systems ». In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. (Seattle, Washington, USA), p. 1909-1916. 1144315: ACM.

Tahvildar, L., and K. Kontogiannis. 2004. « Improving design quality using meta-pattern transformations: a metric-based approach ». *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 16, n° 4-5, p. 331-361.

Takagi, H. 2001. « Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation ». *Proceedings of the IEEE,* vol. 89, n° 9, p. 1275-1296.

Tiberghien, A., N. Moha and A. F. Le Meur. 2007. *Détection semi-automatique des patrons de mauvaise conception dans les architectures orientées objet*. université des Sciences et Technologies de Lille, 40 p.

Travassos, G., F. Shull, M. Fredericks and R. V. Basili. 1999. « Detecting defects in object-oriented designs: using reading techniques to increase software quality ». *SIGPLAN Not.,* vol. 34, n° 10, p. 47-56.

Van Der Straeten R., and M. D'Hondt. 2006. « Model refactorings through rule-based inconsistency resolution ». In *Proceedings of the 2006 ACM symposium on Applied computing*. (Dijon, France), p. 1210-1217. 1141564: ACM.

Van Kempen, M., M. Chaudron, K. Derrick and B. Andrew. 2005. « Towards proving preservation of behaviour of refactoring of UML models ». In *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (SAICSIT)* (South African Institute for Computer Scientists and Information Technologists, Republic of South Africa), p. 252-259.

Wilcoxon, F. 1945. « Individual Comparisons by Ranking Methods ». *Biometrics Bulletin,* vol. 1, n^{o} 96, p. 80-83.

Zhang, J., Y. Lin and J. Gray. 2005. « Generic and Domain-Specific Model Refactoring using a Model Transformation Engine ». *Model-driven Software Development – Research and Practice in Software Engineering,* vol. 2, p. 199-217.