Automated Refactoring using Design Differencing

Iman Hemati Moghadam
School of Computer Science and Informatics
University College Dublin, Ireland
Email: Iman.Hemati-Moghadam@ucdconnect.ie

Mel Ó Cinnéide
School of Computer Science and Informatics
University College Dublin, Ireland
Email: mel.ocinneide@ucd.ie

***Abstract*—Software systems that undergo repeated addition of functionality commonly suffer a loss of quality in their underlying designs, termed *design erosion*. This leads to the maintenance of a system becoming increasingly difficult and time-consuming during its lifetime. Refactoring can reduce the effects of design erosion, but this process requires significant effort on the part of the maintenance programmer. Research into automated refactoring has had some success in reducing the effort involved, however source code refactoring uses refactoring steps that are too small to effect major design changes. Design-level refactoring is also possible, but these approaches operate on design models and do little to help in the subsequent refactoring of the source code.**

In this paper, we present a novel refactoring approach that refactors a program based both on its *desired design* and on its source code. The maintenance programmer first creates a desired design (a UML class model) for the software based on the current software design and their understanding of how it may be required to evolve. Then, the source code is refactored using the desired design as a target. This resulting source code has the same behavior as the original, but its design more closely correlates to the desired design.

We conducted an investigation using several open source Java applications to determine how precisely it is possible to refactor program source code to a desired design. Our findings were that the original program could be refactored to the desired design with an accuracy of over 90%, hence demonstrating the viability of automated refactoring using design differencing.

***Keywords*-Search-based refactoring; automated design improvement; design differencing; refactoring tool**

I. Introduction

A developer frequently faces the problem of extending an existing program with new functionality. Sometimes the new functionality can be added easily, but often the current design of the program does not permit easy extension with the new functionality. This is a critical point in the software lifecycle. If the developer ignores the inadequacy of the current design and simply extends the program with the new functionality, the process of design erosion has started. An eroded design results a low quality system in terms of extensibility, flexibility, and maintainability [1].

The recommended approach is to refactor the system so that its design supports the new extension easily. The developer will study the current program design and decide on a new desirable design that will facilitate the extension of the program with the new functionality. The issue the developer then faces is this: how best to refactor the program to this new, desirable design?

Current industrial refactoring tools provide an implementation of a variety of individual refactorings, but deciding on the exact refactoring sequence required to reach the desired design, and the execution of this sequence, is left to the developer. This is a non-trivial task as evidenced by the reluctance of many development teams to engage in radical design overhaul [2]. At best, performing a radical design overhaul will be time-consuming. It may also prove very difficult to refactor the code to the desired design, or the refactoring may have unexpected adverse side effects on the program design [3].

There are two principal areas of research that are relevant to the problem of design overhaul. Firstly, Search-Based Software Engineering [4] techniques have been used to automatically apply refactorings to source code in order to improve the code based on a set of software quality metrics [3], [5], [6], [7], [8]. While these approaches have been shown to effect source code improvements, the refactoring steps they take are too small to effect radical design improvement.

Secondly, automated improvement of software design, where the design is expressed as a UML model, is another area of research [9], [10], [11]. These approaches can effect radical design improvement, as they are not burdened with the problem of handling source code details. However, while they can help the developer in finding a better design for the program, they do not help in the critical process of refactoring the source code itself.

In this paper we propose a novel approach to refactoring a program to comply with a new design. We assume that the developer has created a new, UML-based desired design for the program they are working with. Our approach then extracts a design from the current program and compares this with the desired design. The result of this detection phase is a set of differences between the current program design and the desired program design. These design differences are then mapped to design-level refactorings. During the subsequent reification phase these design-level refactorings are mapped to source-level refactorings which are then applied to the source code. In general, the order in which the source-level refactorings are performed is critical, and we use a search technique to find a good, though not optimal, ordering.

The viability of this approach depends crucially on how precisely it is possible to automatically refactor source code to conform more closely to a given desired design using the proposed approach. To investigate this further,

we constructed an experiment where we randomly apply several hundred refactorings to a high-quality software application in order to produce a behaviorally-equivalent version of the application, but of poor design quality. This poor quality application is then used as the starting program, and we attempt to recreate the original application using automated refactoring with design differencing. A design differencing tool, JDEvAn [12], is used to extract design models from both programs and produce a set of design differences. From these design differences a sequence of source-level refactorings is generated and, using the refactoring framework Code-Imp [6], applied to the poor quality application. The efficacy of our approach is then assessed in terms of how well this refactoring sequence is capable of regenerating the original application.

The experiment was run on 6 open source Java applications, and it was found that on average over 92% of the refactoring sequence could be applied. These results demonstrate the efficacy of the approach, where the original program designs were rebuilt with a high degree of accuracy.

In summary, the main contributions of this paper are as follows:

- A novel, search-based approach for refactoring a software system based on both its design and source code is presented.
- The efficacy of the proposed approach is assessed using 6 open-source applications.

The remainder of this paper is structured as follows: A brief description of the software tools upon which our approach is based, JDEvAn and Code-Imp, is presented in Section II. The algorithm for refactoring to a desired design is detailed in Section III. We describe our experimental methodology in Section IV, and then report and discuss the results obtained in Section V. A survey of related work is presented in Section VI. Finally, we discuss future plans and conclude in Section VII.

II. Tools Used

In the investigations described this paper, we make use of two software tools, JDEvAn and Code-Imp. A short description of each is presented in the following subsections. The interested reader is referred to [13] and [14] for more information.

A. JDEvAn

JDEvAn (Java Design Evolution and Analysis) [12] is an Eclipse plug-in developed at the University of Alberta. It analyzes a software system's design-evolution history and provides information about the system's history. The version used in this paper contains a Java fact extractor, a query-based change-pattern detection module, and a design differencing algorithm named *UMLDiff*.

The fact extractor is implemented based on the Eclipse Java DOM/AST model. It extracts a UML logical design model from Java source code and stores the extracted information in a PostgreSQL relational database. UMLDiff is a heuristic algorithm that uses lexical and structural similarity to automatically recover differences between one version of a system and the next [14]. JDEvAn's refactoring-detection module defines a suite of queries that attempt to categorize detected differences as refactoring instances [15].

As a brief description of its process, JDEvAn initially extracts two UML models from the source code corresponding to two versions of a Java system. Then, using UMLDiff, the two models are compared and the differences between them are detected. In the final step, the detected differences are categorized, where possible, as design-level refactoring instances.

B. Code-Imp

Code-Imp (Combinatorial Optimisation for Design Improvement) is a fully automated refactoring framework developed by the authors [6], [13], [16] in order to facilitate experimentation in automatically improving the design of existing programs. It takes Java version 6 source code as input and produces as output a refactored version of the program. Its output also comprises applied refactorings and metrics information gathered during the refactoring process. There are three aspects to the refactoring that takes place:

- the set of refactorings that can be applied;
- the type of search technique employed;
- the fitness function that directs the search.

Code-Imp currently supports 14 design-level refactorings categorized into three groups according to their scope as shown in Table I. These are roughly based on refactorings from Fowler's catalogue [17] though they differ somewhat in the details.

The refactoring process is driven by a search technique, e.g. hill-climbing or simulated annealing. The simplest search technique is steepest-ascent hill-climbing, where the next refactoring to be applied is the one that produces the best improvement in the fitness function.

The fitness function is a measure of how "good" the program is, so the fitness function used depends on the quality that we are trying to improve. Code-Imp is normally used to improve software design, so in this case the fitness function is a combination of software quality metrics. Over 40 software quality metrics have been implemented in Code-Imp and can be combined using either a weighted-sum approach or Pareto optimality [5].

As described later in section III, Code-Imp is used in a somewhat different way in this paper. We know the set of refactorings to be applied, but because of dependencies and conflicts between the refactorings the order in which they are applied is critical. The fitness function we use in this work therefore is a measure of the number of refactorings that can be legally applied to the program. For $N > M$, a refactoring that leads to a program where N refactorings can be applied is preferred over one that leads to a program where M refactorings can be applied, and this forms the basis for the fitness function that guides the search.

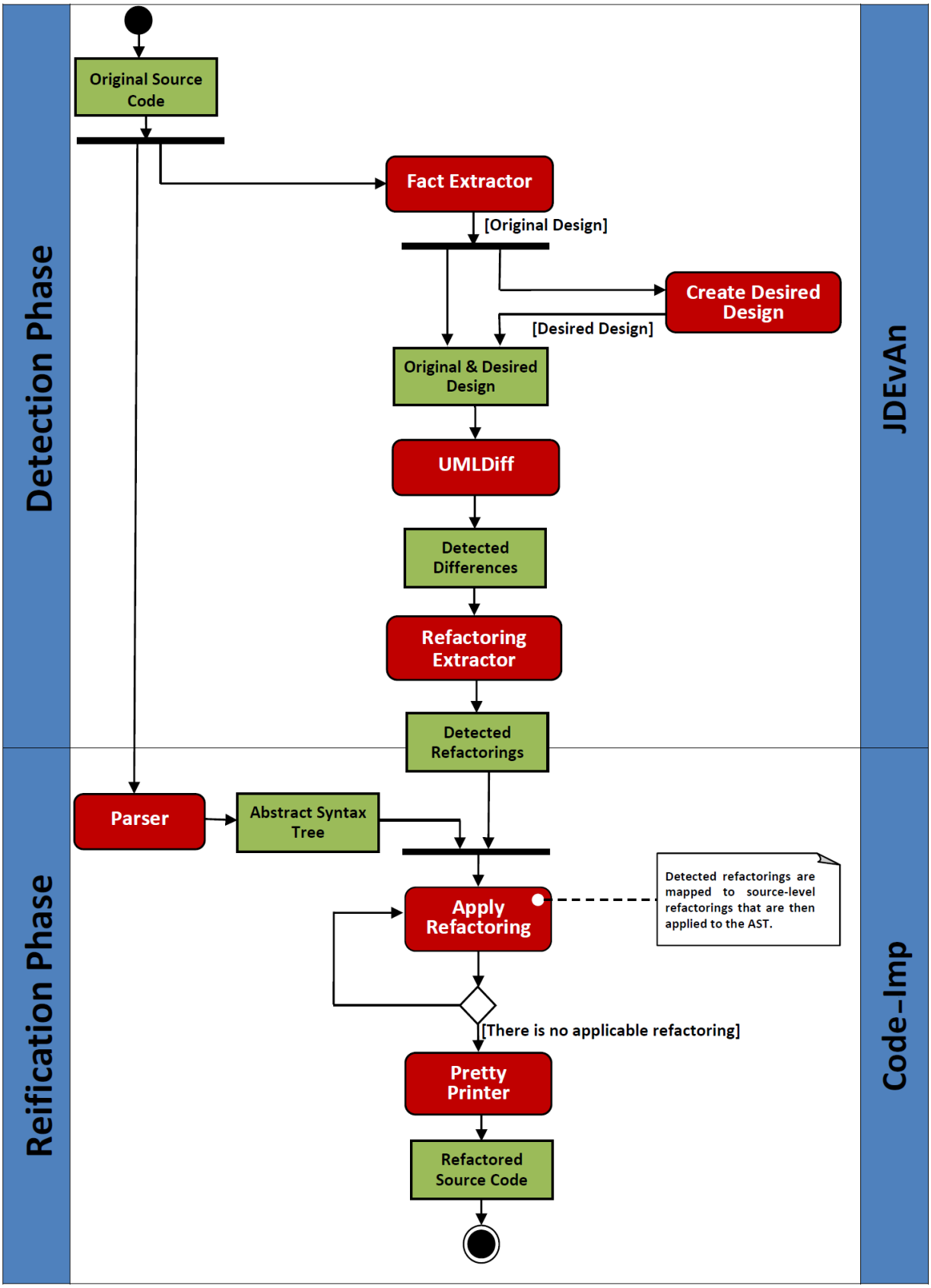

Figure 1. Overview of Automated Refactoring using Design Differencing

Table I
A LIST OF IMPLEMENTED REFACTORINGS IN CODE-IMP

No.	Class-Level Refactorings	Description
1	Extract Hierarchy	Adds a new subclass to a non-leaf class C in an inheritance hierarchy.
2	Collapse Hierarchy	Removes a non-leaf class from an inheritance hierarchy.
3	Make Superclass Concrete	Removes the explicit *abstract* declaration of an abstract class without abstract methods.
4	Make Superclass Abstract	Declares a constructorless class explicitly abstract.
5	Replace Inheritance with Delegation	Replaces a direct inheritance relationship with a delegation relationship.
6	Replace Delegation with Inheritance	Replaces a delegation relationship with a direct inheritance relationship.
	Method-Level Refactorings	
7	Push Down Method	Moves a method from a class to those subclasses that require it.
8	Pull Up Method	Moves a method from some class(es) to the immediate superclass.
9	Decrease Method Accessibility	Decreases the accessibility of a method from protected to private or from public to protected.
10	Increase Method Accessibility	Increases the accessibility of a method from protected to public or from private to protected.
	Field-Level Refactorings	
11	Push Down Field	Moves a field from a class to those subclasses that require it.
12	Pull Up Field	Moves a field from some class(es) to the immediate superclass.
13	Decrease Field Accessibility	Decreases the accessibility of a field from protected to private or from public to protected.
14	Increase Field Accessibility	Increases the accessibility of a field from protected to public or from private to protected.

III. REFACTORING TO A DESIRED DESIGN

In this section the algorithm we have developed to automatically refactor source code towards a desired design is described in detail. By way of preliminaries, some terms that are widely used in this paper are defined:

- **Original source code:** Current version of program source code that is to be refactored.
- **Original design:** Design model extracted from the *original source code*. The model is expressed according to the semantics of the UML meta-model [18].
- **Desired design:** UML design model extracted from the *original source code*, but updated by the developer depending on their understanding of how the system may be required to evolve in the future.
- **Refactored source code:** A version of the original source code that has been refactored to conform more closely to the *desired design*.
- **Detected differences:** The structural differences between the original and desired designs in terms of (a) additions, removals, and moves of model elements in interfaces, classes, attributes and operations and (b) changes in the relationships between these model elements.
- **Detected refactorings:** Detected differences that are categorized as design-level refactoring instances. As a simple example, a method which is moved from a class in the original design to its subclasses in the desired design is detected as a number of move method differences, but as one *Push Down Method* refactoring.
- **Source-level refactoring:** A refactoring that is low-level enough to be applied directly to source code by Code-Imp. Detected refactorings are mapped to source-level refactorings during the reification phase.

A. Algorithm to refactor to desired design

Fig. 1 shows an overview of the set of steps that are followed in the refactoring of the original source code to a new, desired design. Initially, a UML class model is extracted from the *original* source code. This model is named the *original design* and is used without any changes. A second model, named the *desired design*, is an updated version of the original design that has been changed by the maintenance programmer based on their understanding of how the system is to be extended with new functionality. In the next step, the two models are automatically compared using a differencing algorithm. The result of this stage is a set of *detected differences* which are stored in a database. The detected differences are then automatically categorized as *detected refactorings* using a suite of predefined queries. Finally, the *original source code* is refactored using a heuristic approach based on the *detected refactorings* in order to conform more closely to the *desired design*. Note that this paper is not concerned with activities involved in creating a desired design from the original one. Our focus is primarily on how precisely it is possible to refactor the original source code towards its desired design.

As shown in Fig. 1, the algorithm is divided into a *detection phase* and a *reification phase*. The detection phase involves activities related to detecting structural differences between two versions of the program and expressing the differences as refactoring instances. This phase is implemented using JDEvAn. The reification phase includes activities related to refactoring the original program source code based on the detected refactorings. This stage is implemented using Code-Imp. A detailed description of the implemented algorithm based on these two phases is as follows:

1) ***Detection Phase****:* JDEvAn's fact extractor recovers a model of the class design of the original source code based on the semantics of the UML model [15]. The desired design is also expressed in terms of the UML model. Both the original design and the desired design are stored in a PostgreSQL relational database. The two models are then compared with each other using a structural UML differencing algorithm called *UMLDiff* [14]. UMLDiff uses lexical and structural similarity to compare the two models. It results a set of differences detected between the two designs, which are also saved in the database. In the final step, the recovered differences are categorized, where possible, as refactoring instances using certain predefined

queries [15]. In this paper, we used 8 implemented queries related to 14 source-level refactorings implemented in Code-Imp as shown in Table I. In two cases, *Replace Inheritance with Delegation* and vice versa, we used two separate queries. However, in other cases, a query retrieves information related to two refactorings. For example, a *Pull Up* query, retrieves information related to two type of refactorings namely *Pull Up Method* and *Pull Up Field* refactorings.

As an example of an implemented query, an *Extract Hierarchy* refactoring is recognized if (1) a new class is added to the system, (2) the new class inherits from at least one class in the system other than *Java.lang.object*, and (3) the superclass of one or more classes is changed to the new added class.

*2) **Reification Phase**:* In this phase the original program source code is refactored based on the detected refactorings in order to become as close as possible to the desired design. However, the source code as a plain textual form is not rich enough to be used as a basis for refactoring [19]. Therefore, as first step, an initial abstract syntax tree is extracted from the source code. The abstract syntax tree is created from both syntactic and semantic analyses and contains all the information that is required during the entire transformation process.

Detected refactorings are mapped to source-level refactorings and applied to the abstract syntax tree in a specific order determined by the search algorithm in use. A refactoring is accepted if (1) its preconditions are true and (2) it complies with the demands of the search technique in use. However, during precondition checking, it is possible that a detected refactoring cannot be mapped easily to the source-level refactorings, or it cannot be executed because of a failing precondition. In this case, Code-Imp tries to find an equivalent sequence of refactorings (1) that is applicable to the existing source code, and (2) whose execution has the same effect as candidate refactoring. If Code-Imp can find an equivalent sequence of refactorings, the candidate refactoring is replaced with that sequence, and the process is continued with a new detected refactoring.

The refactoring process is repeated until all the detected refactorings have been processed, or there is no refactoring remaining that fulfills the requirements of the search technique. After applying the last possible refactoring to the system, the abstract syntax tree is pretty printed to the source code files.

B. Reification Phase in more detail

In the previous section a high-level description of the reification phase was provided, with much of the detail omitted. In this section a more detailed description of this phase is presented.

During the reification phase it is possible that a detected refactoring cannot be mapped directly to a source-level refactoring that can be executed on the existing source code. The problem is described with a simple example as depicted in Fig. 2. The figure shows two UML class

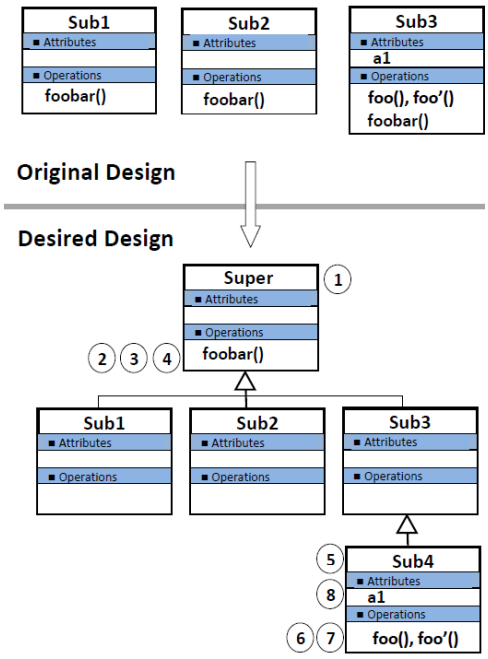

Figure 2. The original and desired UML class diagrams of a simple example program. For simplicity, only methods and fields that are used in the refactoring process are shown.

diagrams, the original and desired designs for a simple program. During the detection phase, JDEvAn detects twelve differences between the two designs and maps them to eight detected refactorings as shown in Fig 2. Two new classes named *Super* and *Sub4* are added to the hierarchy structure using two *Extract Hierarchy* refactorings. Method *foobar()* is moved from *Sub1*, *Sub2* and *Sub3* to *Super* using three *Pull Up Method* refactorings, and two methods *foo()* and *foo′()* along with attribute *a1* are moved from *Sub3* to *Sub4* using two *Push Down Method* and one *Push Down Field* refactorings. In the reification phase, two of these detected refactorings, namely *Extract Hierarchy*, are mapped directly to equivalent source-level refactorings. However, other detected refactorings cannot be applied to the original source code.

As an example, after performing the detected refactoring *Pull Up foobar() from Sub1 to Super*, the other two related refactorings, *Pull Up foobar() from Sub2 and Sub3 to Super*, are rejected during precondition checking. It happens because there is a similar method in the destination class, and a precondition of the *Pull Up Method* refactoring is that the method must not override any method in the destination class. Replacing these three detected refactorings with one *Pull Up Method* refactoring as a source-level refactoring is our implemented solution to this problem. Therefore, all three *foobar()* methods are moved to their immediate superclass using one newly-created *Pull Up Method* refactoring.

As another example, not depicted in Fig 2, consider a method that has been moved to some of its subclasses in the desired design. The detection phase returns a number of *Push Down Method* refactorings depending on the number of subclasses. However, during the reification

phase, none of these refactorings can be applied to the system. Indeed, all these refactorings are rejected during precondition checking because as soon as the method is moved to one of its subclassses, its reference in other subclasses is no longer valid. Our implemented solution to this case is to replace all these detected *Push Down Method* refactorings with one single source-level refactoring, *Push Down Method*.

Apart from cases where detected refactorings must be replaced with a number of source-level refactorings, there are situations where it is necessary to add some new refactorings to the detected refactoring set. The problem is described using Fig 2. During the reification phase, neither the private methods *foo()* and *foo′()* nor the private attribute *a1* can be moved to their subclass because both methods use attribute *a1*, and also method *foo()* calls method *foo′()*. Hence, as soon as one of these elements is moved to the subclass, its reference in the superclass is no longer valid, leading to a compiler error. Therefore, these refactorings are all rejected during precondition checking[1].

In this case, our solution is to add new refactorings to the detected refactoring set. Initially, two new refactorings, *Increase Field Accessibility* and *Increase Method Accessibility*, are added to the refactoring set to increase the accessibility of attribute *a1* and method *foo()* to *protected*. Then the detected move refactorings are performed on the system. After this, the accessibility of the attribute and the method are brought back to their original state using a number *Decrease Accessibility* refactorings. The accessibility of the attribute and the method are changed to *private* using a new *Decrease Field Accessibility* and a new *Decrease Method Accessibility* refactorings respectively. Note that the newly-created refactorings must be applied to the system in some specific order. Also, when a detected refactoring is replaced with a number of source-level refactorings, those refactorings must all be performed to the system. Otherwise, their effects on the system must be canceled.

In summary, during the reification phase, when an individual refactoring is selected to be applied to the system, Code-Imp tries to find other necessary refactorings and applies them to the system in a behavior-preserving order. This process may involve replacement of one or a group of related refactorings with some equivalent source-level refactorings, or the addition of some new refactorings to the related refactorings. However, note that in this stage, Code-Imp does not aim to find a behavior preserving order between all detected refactorings. A search-based technique is used to decide on a suitable order in which to apply the refactorings.

During the reification phase other problem faced is that we have a set of N source-level refactorings to be applied, but due to interdependencies between these refactorings only certain sequences are valid. Finding a sequence of refactorings such that the preconditions for each refactoring in the sequence is valid after applying the previous ones is not an easy task. What makes the process more difficult is that the effect of each refactoring is only seen after applying it to the source code. Therefore, it is possible that a refactoring changes the behavior of the system at its turn while it satisfied the preconditions at the start of the sequence.

In this paper, we expressed the task of finding the best sequence of refactorings as a search-based problem. Our search problem then consists in finding a sequence of source-level refactorings that results the desired program design. The fitness function used in this work is a measure of the number of refactorings that can be legally applied to the program. For $N > M$, a refactoring that leads to a program where N refactorings can be applied is preferred over one that leads to a program where M refactorings can be applied, and this forms the basis for the fitness function that guides the search. In this way, the fitness function promotes refactorings that increase the number of refactorings that can be applied in the next step. Note that the desired design is indeed achieved after applying all detected refactorings. Therefore, the fitness function leads the current design to a state that may have more potential to reach the desired design. During the reification phase the value of the fitness function may fall as well as rise. However, the search is guaranteed to make progress because once a refactoring is applied to the system, it is deleted from the detected refactoring list.

IV. Experimental Design

The efficiency of refactoring by design differencing, depends crucially on how *precisely* it is possible to refactor an original program source code towards a desired design. To investigate this, we conducted experiments using 6 open-source Java applications. Table II shows summary information about the applications used in this study.

Table II
Software applications used in this study

No.	System	Description	LOC	#Classes
1	JHotDraw 5.3	Graphics	15,254	208
2	JGraphX 1.5.1.6	Java Graphing	49,661	229
3	JTar 1.2	Compression	9,010	59
4	HtmlUnit 1.4	Unit-test framework	12,297	194
5	GanttProject 2.0.9	Scheduling	43,579	547
6	XOM 1.2.6	XML API	25,538	217

However, the initial design in each experiment is created differently from what has been described in Section III. In each experiment, the application under investigation was initially, randomly and massively refactored to change its design structure. As expected, this led to a decrease in the quality of the design of the software. Then, using the approach described in Section III, we tried to rebuild the high-quality original source code from its low-quality refactored version based on detected design differences between two versions. Therefore, in our experiments, the original program design is in fact the desired design, and the degraded program design is the original design.

[1]During the experiments for this paper, we observed a strong interdependence between the detected refactorings. Indeed the majority of detected refactorings were rejected at code level because of their interdependence as in the described examples.

Table III
EVALUATION DATA SUMMARY

No.	System	Random refactorings	Detected refactorings	Source-level refactorings	Inapplicable refactorings	Correctly applied refactorings
1	JHotDraw 5.3	500	414	483	14	97%
2	JHotDraw 5.3	500	426	516	27	95%
3	GanttProject 2.0.9	500	491	529	85	84%
4	JTar 1.2	200	107	127	9	93%
5	Xom1.2.6	500	341	449	46	90%
6	HtmlUnit 1.4	500	374	466	23	95%
7	JGraphX 1.5.1.6	500	410	454	46	90%

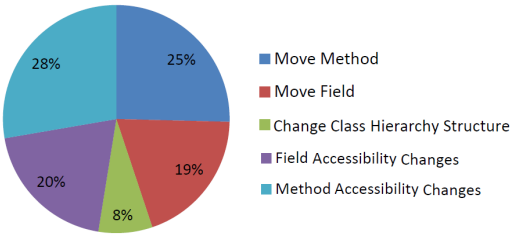

Figure 3. Breakdown of the random refactorings applied across all the experiments

V. RESULTS AND EVALUATION

In this section we present our evaluation of the proposed approach. The main aim is to investigate how precisely it is possible to refactor source code. However, at first, we need to create initial design as described in Section IV. Thus, initially, 500 random refactorings are applied to each application.[2] The random refactoring process is completely automated and is performed by Code-Imp. Fig. 3 presents an overview of the refactorings applied across all the experiments. As illustrated, fourteen design-level refactorings described in Table I are classified into 5 groups according to their scope.

In the next step, the randomly refactored version is refactored towards its desired design. Table III shows a summary of each experiment as follows: Column 3 shows the number of random refactorings applied to the original source code. Column 4 presents the number of detected refactorings recovered during detection phase, and column 5 shows the number of source-level refactorings that were applied during the reification phase.[3] Column 6 represents the number of refactorings that were not applied to the system during the reification phase. It shows differences between the achieved and desired designs in terms of source-level refactorings. Finally, the last column includes percentage of correct applied refactorings in total.

The average of applied refactorings indicates that around 92% of initial random refactorings were revoked in reification phase. It means that the original program designs were rebuilt with a high degree of accuracy, hence demonstrating the capability of our design differencing approach to refactor source code to a desired design. However, to be certain that inapplicable refactorings are only differences between a desired design and the actual achieved design, we compared all desired designs with their refactored version using UMLDiff. We also manually compared desired and achieved designs in two experiments. We manually compared *JHotDraw*'s designs in experiment 2, and *JTar*'s designs in experiment 4. *JTar* was selected because it has the biggest differences between random refactorings and source-level reafctorings. On the other hand, *JHotDraw* was selected because the number of source-level refactoring was more than random refactorings in the second experiment. The results confirmed that the inapplicable refactorings are indeed the only differences between designs.

We also measured some software quality metrics during both random refactoring and reification phases. Fig. 4 shows how two of these metrics namely *CAMC* [20] and *SCC*[4] [21], changed in the first experiment. As illustrated, the quality in terms of cohesion is gradually degraded for both metrics during the random refactoring phase. However, because the process is completely random, in some points refactorings had positive impact on the design quality as illustrated in the figure. During reification phase, both metrics are progressively improved, and as expected, there are some minor differences in the final metrics values between the original source code as *desired design* and the refactored version as *final design*.

The results were also manually inspected in two experiments to find the reason why some refactorings could not be applied to the system during reification phase. The investigation showed that some of these refactorings were not in fact detected during detection phase. This problem mostly happened when a hierarchy structure was changed radically using refactorings categorized as *change class hierarchy structure*.[5] These changes reduce parent similarity between method/field move candidates that their declaring classes are related through inheritance. A move method/field change cannot be detected because structure similarity between the candidates is less than a move

[2]The number of random refactorings is reduced to 200 refactorings in experiment 4. To prevent duplication in random refactorings, the refactoring process is stopped as soon as the number of duplicate refactorings is more than 10% of applied refactorings.

[3]As mentioned, a detected refactoring can be replaced with a sequence of source-level refactorings, and also a group of detected refactorings can be replaced with one source-level refactoring. Hence, the number of source-level refactorings can be more or less than the number of random refactorings.

[4]*CAMC* is an abbreviation of *cohesion among methods of class*, and *SCC* is an abbreviation of *similarity-based class cohesion*. Both are defined as high-level design quality metrics, and an increase in their value means an improvement in program cohesion.

[5]It includes four refactorings namely *Extract Hierarchy*, *Collapse Hierarchy*, *Replace Inheritance with Delegation* and vice versa.

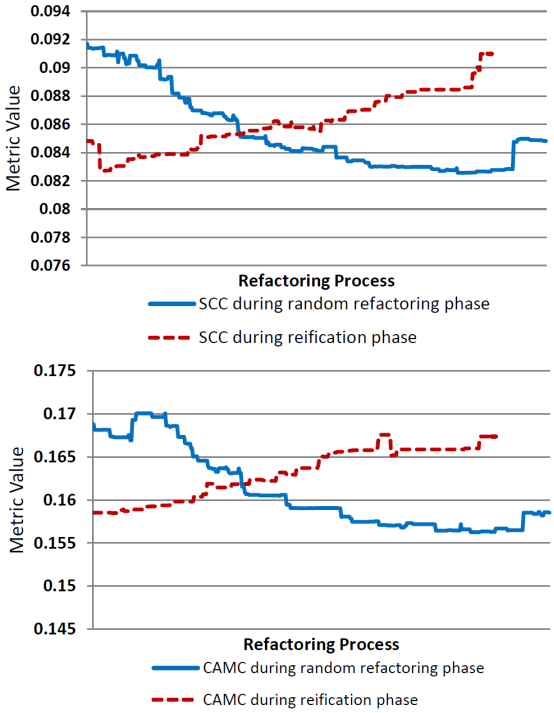

Figure 4. Graphs of metrics changes during random refactoring and reification phases

threshold determined in UMLDiff algorithm. A reduction in the value of this threshold can alleviate this problem. However, it can also increases the number of false positive detected refactorings in the detection phase.

In summary, while the results were negatively affected by restrictions associated with the differencing tool, a high degree of accuracy in terms of number of refactorings applied to the system as well as metrics values, shows the capability of the approach to refactor source code to its desired design.

VI. Related Work

Work related to this paper can be divided into two areas: automated design improvement through means of (1) Code-level Refactoring and (2) Design-level Refactoring.

In the field of automated design improvement, several research works have focused on semi-automated approaches which are in-between the manual and full-automated process.

Tsantalis and Chatzigeorgiou [22] propose a semi-automatic approach for identifying refactoring opportunities related to code smells based on system's history. In this approach, a detected refactoring that occurs in highly changeable code fragment has a higher priority for refactoring. In other work [23], they propose an approach for detecting move method refactoring opportunities based on code smells. Tsantalis and Chatzigeorgiou [24] also propose an approach to extract refactoring opportunities that introduce polymorphism as a solution to state-checking problems. While in the mentioned approaches, refactoring opportunities are ranked based on their effect on design quality, it is the designer responsibility to take restructuring decision that a refactoring should be performed to the system or not. In another semi-automatic approach, Marinescu et al. [25] develop an interactive tool called *inCode* for identifying four kinds of design flaws, with simultaneous contextual refactoring hints for their removal. The design problems are related to improper distribution of class elements, and are detected using a quality assessment module.

In contrast to the aforementioned semi-automatic approaches, there are other research works aimed to automate the whole process. There are several works that focused on automated the process using search-based techniques.

O'Keeffe and Ó Cinnéide [6] propose an automated design improvement approach to improve the quality of program in terms of *flexibility*, *understandability*, and *reusability* based on the QMOOD quality model [26]. They conducted some experiments on 2 medium-sized Java applications to investigate if automated search-based refactoring could improve a program's design. They report a significant and minimal improvement on understandability and flexibility respectively. However, they find the QMOOD reusability function unsuitable.

In other work, O'Keeffe and Ó Cinnéide [27] investigate if a program can be automatically refactored to reduce its dissimilarity based on a set of design metrics extracted from an example program. The approach can be used to improve the design quality when a sample program has some desirable features, such as ease of maintenance, and it is desired to achieve theses features. They carried out some experiments on three Java programs in a way that each program was refactored based on metrics values extracted of other two programs. They report the ability of the approach in reducing dissimilarity between programs.

Seng et al. [3] propose an approach for improving design quality by moving methods between classes in a program. They use a genetic algorithm with a fitness function based on coupling, cohesion, complexity and stability to produce a desirable sequence of move method refactorings. Improving design quality by moving methods among classes was also investigated by Harman and Tratt [5], except they introduce and use a Pareto optimality approach to make combination of metrics easier.

Jensen and Cheng [7] use refactoring in a genetic programming environment to improve the quality of the design in terms of a quality model. The approach was capable of introducing design pattern through the refactoring process, which helps to change the design radically.

Apart from the aforementioned approaches, which perform refactoring on source code, there has been also some interest to automate refactoring process at design-level. For example, Simons et. al [11] show how an interactive, evolutionary search technique can improve activities in upstream software design. They use a multi-objective genetic algorithm to design a class structure from a design problem derived from use cases. In this approach, a designer and software agents cooperate together to guide the search towards a better design.

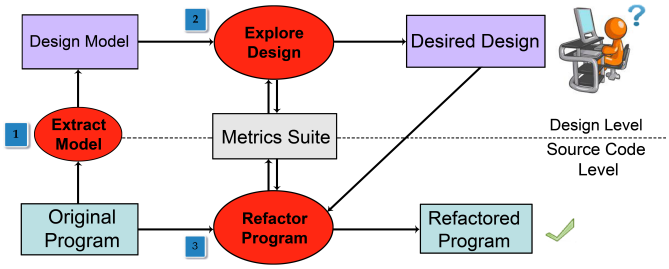

Figure 5. Multi-level refactoring using a search-based refactoring approach [28]

VII. Conclusion and Future Work

Refactoring is a proven solution to improve software quality, and hence facilitate maintenance activities. However, refactoring is not a straightforward activity, which is a motivation for automated tool support. In this paper, we have presented a novel approach that helps to reduce upcoming maintenance efforts by automating the design improvement process. In our approach, program source code is automatically refactored towards a high-quality desired design. The desired design is an improved version of the current system design, and is developed by a designer familiar with the system.

The principal benefit of the approach is that it enables automated refactoring towards a high-quality desired design, and hence improve maintenance productivity. We have demonstrated the capability of the approach using 6 open source examples. The main aim was to investigate how precisely it is possible to automatically refactor a program source code towards its desired design. The efficacy of the approach was proven by a high degree of accuracy which was achieved in the experiments.

The main aim of our future work in this area is the automation of the entire design improvement process. The process of creating a new high-quality design from an initial one is still a non-trivial task. Our proposal is based on a multi-level refactoring approach to facilitate both design- and code-level refactoring activities [28]. A multi-level refactoring approach is a combination of *code* and *design improvement* tools. In this approach, the refactoring process is divided into two steps, *design exploration* and *code refactoring*, as illustrated in Fig. 5. As a brief description of the process, initially, the program design is extracted from the source code in terms of a UML design model. Then, in a design exploration phase, the extracted design is transformed to a better one in terms of a *metrics suite* as well as the *user perspective*. The source code is then refactored based on both the *improved design* and the *metrics suite*. The result of this approach is high-quality source code whose design has been approved by the developer. In this approach, because design exploration and code refactoring are both automated processes, the refactoring process is completely straightforward from user perspective.

Acknowledgment

This work was funded by the Irish Programme for Research in Third-Level Institutions (PRTLI) as part of the Lero Graduate School in Software Engineering (www.lero.ie).

References

[1] J. van Gurp, S. Brinkkemper, and J. Bosch, "Design preservation over subsequent releases of a software product: a case study of baan ERP: Practice articles," *Journal of Software Maintenance and Evolution*, vol. 17, pp. 277–306, July 2005.

[2] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 287–297.

[3] O. Seng, J. Stammel, and D. Burkhart, "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*, ser. GECCO '06. New York, NY, USA: ACM, 2006, pp. 1909–1916.

[4] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," Tech. Rep. TR-09-03, April 2009.

[5] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level," in *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07, 2007, pp. 1106–1113.

[6] M. O'Keeffe and M. Ó Cinnéide, "Search-based refactoring for software maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[7] A. C. Jensen and B. H. Cheng, "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*, 2010, pp. 1341–1348.

[8] H. Kilic, E. Koc, and I. Cereci, "Search-based parallel refactoring using population-based direct approaches," in *Proceedings of the 3rd international Conference on Search Based Software Engineering*, ser. SSBSE'11, 2011, pp. 271–272.

[9] M. Amoui, S. Mirarab, S. Ansari, and C. Lucas, "A genetic algorithm approach to design evolution using design pattern transformation," *International Journal of Information Technology and Intelligent Computing*, vol. 1, no. 2, pp. 235–244, June 2006.

[10] O. Raiha, K. Koskimies, and E. Makinen, "Genetic synthesis of software architecture," in *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL '08)*. Melbourne, Australia: Springer, 2008, pp. 565–574.

[11] C. Simons, I. Parmee, and R. Gwynllyw, "Interactive, evolutionary search in upstream object-oriented class design," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 798–816, Nov 2010.

[12] "JDEvAn," 2008, http://webdocs.cs.ualberta.ca/∼stroulia/Zhenchang_Xing_Old_Home.

[13] I. Hemati Moghadam and M. Ó Cinnéide, "Code-Imp: a tool for automated search-based refactoring," in *Proceeding of the 4th workshop on Refactoring tools*, ser. WRT '11. New York, NY, USA: ACM, 2011, pp. 41–44.

[14] Z. Xing and E. Stroulia, "Differencing logical UML models," *Journal of Automated Software Engineering.*, vol. 14, pp. 215–259, June 2007.

[15] ——, "Refactoring detection based on UMLDiff change-facts queries," in *Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 263–274.

[16] M. O'Keeffe and M. Ó Cinnéide, "Search-based refactoring: an empirical study," *Journal of Software Maintenance and Evolution*, vol. 20, no. 5, pp. 345–364, 2008.

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

[18] OMG(2003), "Unified Modeling Language Specification, formal/03-03-01, Version 1.5," http://www.omg.org.

[19] W. F. Opdyke, "Refactoring: A program restructuring aid in designinig object-oriented application frameworks," Ph.D. dissertation, Department of Computer Science, Champaign, IL, USA, 1992.

[20] J. Bansiya, L. Etzkorn, C. Davis, and W. Li, "A class cohesion metric for object-oriented designs," *Journal of Object Oriented Programming*, vol. 11, no. 08, pp. 47–52, 1999.

[21] J. Al-Dallal and L. C. Briand, "An object-oriented high-level design-based class cohesion metric," *Information & Software Technology*, vol. 52, no. 12, pp. 1346–1361, 2010.

[22] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, ser. CSMR '11. IEEE Computer Society, March 2011, pp. 25–34.

[23] ——, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, pp. 347–367, May 2009.

[24] ——, "Identification of refactoring opportunities introducing polymorphism," *Journal of Systems and Software*, vol. 83, pp. 391–404, March 2010.

[25] R. Marinescu, G. Ganea, and I. Verebi, "Incode: Continuous quality assessment and improvement," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, ser. CSMR '10, March 2010, pp. 274–275.

[26] J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4–17, Jan 2002.

[27] M. O'Keeffe and M. Ó Cinnéide, "Automated design improvement by example," in *Proceeding of the conference on New Trends in Software Methodologies, Tools and Techniques*, 2007, pp. 315–329.

[28] I. Hemati Moghadam, "Multi-level automated refactoring using design exploration," in *Proceeding of the 3rd International Symposium on Search Based Software Engineering*, ser. SSBSE '11, September 2011, pp. 70–75.