Resolving Conflict and Dependency in Refactoring to a Desired Design

Iman Hemati Moghadam*, Mel Ó Cinnéide**
*Department of Computer Science, University College London, United Kingdom
**School of Computer Science and Informatics, University College Dublin, Ireland
I.moghadam@ucl.ac.uk, mel.ocinneide@ucd.ie

Abstract

Refactoring is performed to improve software quality while leaving the behaviour of the system unchanged. In practice there are many opportunities for refactoring, however, due to conflicts and dependencies between refactorings, only certain orders of refactorings are applicable. Selecting and ordering an appropriate set of refactorings is a challenging task for a developer. We propose a novel automated approach to scheduling refactorings according to their conflicts and dependencies as well as their contribution to design quality expressed by a desired design. The desired design is an improved version of the current program design, and is produced by the developer. Our approach is capable of detecting conflicts and dependencies between refactorings, and uses a sequence alignment algorithm to identify the degree of similarity between two program designs expressed as sequence of characters, thereby measuring the contribution of a refactoring to achieving the desired design. We evaluated our approach on several sample programs and one non-trivial open source application. Our results demonstrate the ability of the approach to order the input refactorings so as to achieve the desired design even in the presence of intense inter-refactoring conflict and dependency, and when applied to a medium-sized, real-world application.

1. Introduction

Refactoring is performed to improve the quality of the software in some way. It may involve floss refactoring, where minor improvements are applied frequently, typically several times a day, or it may involve remedial refactoring[1] where a more significant design overhaul takes place [18]. In this paper we are concerned with automated refactoring support for the remedial refactoring scenario. A developer performing remedial refactoring typically has a notion of a desired design that they are refactoring the program towards. This desired design may come about by way of an interactive design process, as in the work of Simons et al. [24, 25] or it may be created by the intellectual effort of the developer [7, 9]. Either way, the challenge the developer faces is that of refactoring the program from its current design to its new, desired design.

In earlier work, we presented an approach to refactor a program based both on its desired design and on its source code [9]. In this work, a new UML-based desired design is first created by the developer based on the current software design and their understanding of how it may be

[1] Termed 'root canal' refactoring by Murphy-Hill et al. [18] and 'batch mode' refactoring by Liu et al. [15]. Liu et al. [15] provide strong evidence of the practical importance of this type of refactoring.

required to evolve. The resulting design is then compared with the original one using a differencing algorithm [32], and the detected differences are expressed as refactoring instances. The original source code is then refactored using a heuristic approach based on the detected refactorings to conform more closely to the desired design [9]. Overall, the process of refactoring the program to comply with its desired design involves three distinct steps as follows:

1. The developer must decide what refactorings are required to bring the program from its current design to its desired design.
2. They must decide in what order the refactorings should be applied.
3. The refactorings must then be applied to the program in this sequence.

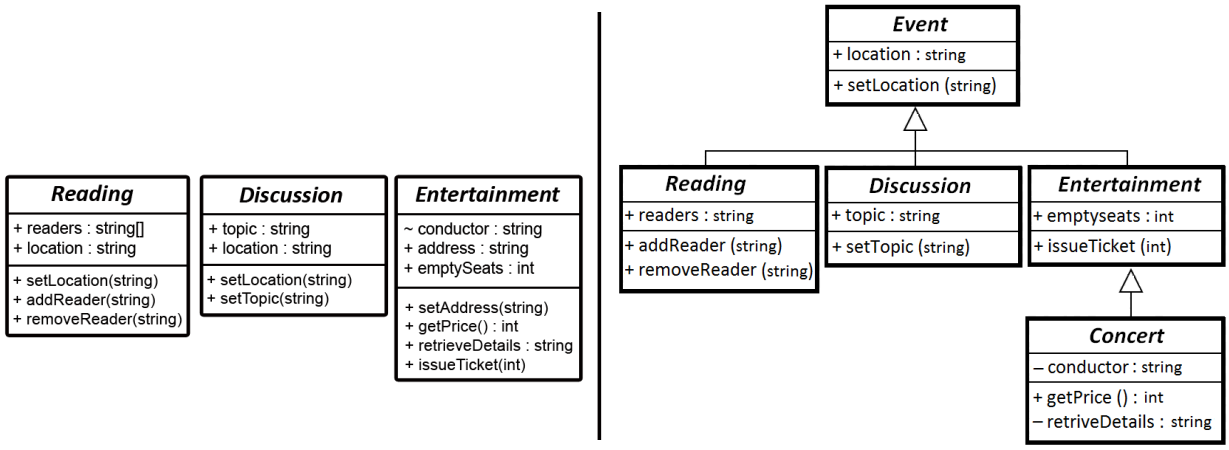
As mentioned, recent work has sought to automate this refactoring process. For example, UML-Diff [31, 32] is a tool that addresses step (1) by detecting what refactorings are required to bring a program design from its current state to a new desired design. Step (3) is supported by a broad range of refactoring tools that apply individual refactorings, such as the Eclipse Refactoring Tools, and also by more sophisticated research prototypes, such as *Code-Imp*, that can apply a series of refactorings guided by a fitness function [8]. The focus of this paper however, is step (2), the ordering of the refactorings into a valid sequence.

Given a set of refactorings, finding a valid sequence in which they may be applied is a non-trivial problem. A refactoring is characterised by a precondition and a postcondition. The precondition determines if the refactoring may be applied, and the postcondition states what the result of applying the refactoring is, assuming its precondition was true when it was applied. A refactoring may be applicable to the initial program, but if it is not applied then, another refactoring in the sequence may render it inapplicable. Conversely, a refactoring may be inapplicable to the initial program, but another refactoring in the sequence may render it applicable later on. These observations have led to the notions of *conflict* and *dependency* in a refactoring sequences [14, 17]. Two refactorings are in *conflict* if they cannot both be applied to the program, e.g. a method cannot be moved to a deleted class. A *dependency* exists between two refactorings if they can only be applied in a particular order, e.g. a refactoring that creates a new class must be executed before a refactoring that moves a method to that new class. Precise definitions of these terms are provided in Section 3.2.

The question addressed in this paper then is as follows. Given a set of proposed refactorings that are to be applied to a program so as to bring its design to a desired state, how can the refactorings be ordered such that they can be applied to the program while respecting the constraints imposed by the conflicts and dependencies that may exist between the refactorings in the set? To answer this question, we propose an automated refactoring scheduling approach that finds a valid order of the refactorings in the set, according to their conflict and dependency relationships as well as their contribution to achieving the desired design. The main contribution of this paper is two-fold:

– We extend the refactoring scheduling algorithm proposed by Liu et al. [14] by considering not just *conflicts* between refactorings but also *dependencies*. Furthermore, we take into account a type of refactoring conflict not handled in the work of Liu at al., where the application of one refactoring violates the precondition of another. We refer to this algorithm as *REDaCT* (REfactoring Dependency and ConflicT).

– We develop the idea of refactoring to a *desired design*, introduced by the authors in earlier work [9], and show how it can be used to guide the refactoring process more effectively. In particular we measure not only the effect of a refactoring in terms of its direct contribution to achieving the desired design, but also its indirect contribution in terms of the refactorings it enables and disables. This extension to the REDaCT algorithm is referred to as *REDaCT+*.



(a) The initial design

(b) The desired design

Figure 1: UML class diagrams of an *Event* application

The remainder of this paper is structured as follows. Section 2 presents a motivating example to illustrate the necessity of scheduling refactorings. In Section 3 we deal with preliminaries by providing a brief description of the software tool upon which our approach is based, *Design-Imp*, and defining precisely our notions of conflict and dependency. The proposed scheduling approach to find a valid order between the set of refactorings according to their conflict and dependency relationships, REDaCT, is explained in Section 4, while in Section 5 the REDaCT+ algorithm is presented which extends REDaCT by considering the direct and indirect contribution of each refactoring to achieving the desired design. In Section 6 the REDaCT and REDaCT+ algorithms are evaluated on a number of examples. A survey of related work is presented in Section 7, while in Section 8 we conclude the paper and provide some suggestions for future work.

2. Motivating Example

Consider as a motivating example, the simplified UML class diagrams shown in Fig. 1. The design in Fig. 1a represents the original design while the design in Fig. 1b represents the desired design that the developer would like the program to have. We applied the design differencing approach proposed by the authors in earlier work [9] that takes as input two UML class diagrams and then uses a UML design differencing algorithm to find differences between the designs and categorises these as refactoring instances. In other words, the approach returns the refactorings that are required to bring a program design from its current state to a new desired design. In this example, the desired design is achieved after applying 15 refactorings to the initial design as follows:

$\mathbf{R_1}$, $\mathbf{R_2}$: Two classes, *Event* and *Concert*, are added to the design using *Extract Hierarchy* and *Extract Subclass* refactorings respectively.

$\mathbf{R_3}$, $\mathbf{R_4}$: Field *address* and method *setAddress*, both in class *Entertainment*, are renamed to *location* and *setLocation* using *Rename Field* and *Method* refactorings respectively. These refactorings prepare the application of refactorings R_7 and R_{10} described below.

$\mathbf{R_5}$, $\mathbf{R_6}$, $\mathbf{R_7}$: The *location* fields in classes *Discussion* and *Reading* and the field *address* in the class *Entertainment* are pulled up to the class *Event* using three separate *Pull Up Field* refactorings.

R_8, R_9, R_{10}: The *setLocation* methods in classes *Discussion* and *Reading* and the method *setAddress* in the class *Entertainment* are pulled up to the class *Event* using three separate *Pull Up Method* refactorings. Refactorings R_5 to R_{10} reduce code duplication and improve readability.

R_{11}, R_{12}, R_{13}: Two methods *getPrice* and *retrieveDetails* as well as the field *conductor*, all defined in the class *Entertainment*, are pushed down to the class *Concert* using two separate *Push Down Method* refactorings and one *Push Down Field* refactoring respectively. The motivation for these refactorings is to move features that are used only in some instances of the original class.

R_{14}, R_{15}: To simplify the interface and improve understandability, the method *retrieveDetails* and the field *conductor* are made more private using the *Decrease Method Accessibility* and *Decrease Field Accessibility* refactorings respectively.

The aim is to find an order between the aforementioned refactorings that, while requiring the minimum effort, results in the desired design from the initial design. However, because of interdependencies between the refactorings, only some specific refactoring orders are applicable to the initial design. A list of interdependencies between the aforementioned refactorings is as below:

- Refactorings R_3 to R_{13} are directly dependent on R_1, or R_2, i.e. a method or field cannot be moved to a class if the target class has not been created yet.
- The *setLocation* methods in the class *Discussion* use the field *location* of the local class. Therefore, *Pull Up setLocation* (R_8) should be applied to the design after *Pull Up location* (R_5). Were the method to be pulled up before the field, it would also be necessary to add an instance of the local class as parameter to the method, resulting in a method with two input parameters, which differs from the corresponding method in the desired design.
- For the same reason, two *Pull Up Method* refactorings, R_9 and R_{10}, should be applied to the design only after their corresponding *Pull Up Field* refactorings, R_6 and R_7, have been applied.
- The refactoring *Rename address* (R_3) can be applied to the design before or after *Pull Up address* (R_5). If R_3 is performed before R_5, then other two *Pull Up Field* refactorings namely R_6 and R_7 can be applied before or after R_3. Otherwise, R_6 and R_7 must be applied to the design after R_3. The second case happens because a precondition in *Rename Field* prevents the field name from being changed if there is already a field with the same name in the class and the fields are used by different methods.
- The method *getPrice* should be pushed down to its subclass using refactoring R_{11} before moving the method *retrieveDetails* and the field *conductor*. This is necessary as *getPrice* uses both this method and field. Were *retrieveDetails* or *conductor* to be pushed down to the subclass (using R_{12} or R_{13}) before *getPrice*, then they would not be accessible in *getPrice*. Similarly, the method *retrieveDetails* should be moved before the field *conductor*.
- The accessibility of the field *conductor* can be reduced using R_{15} after the field is pushed down to the subclass by R_{13}. Were the accessibility of the field to be reduced before the push down refactoring, then the field would not be accessible in the subclass. This would prevent the pushing down of the methods *getPrice* and *retrieveDetails* as well as the field itself to the subclass. A similar situation arises for the *Decrease Accessibility Method* refactoring (R_{14}). The accessibility of the method *retrieveDetails* should be reduced only after the method is pushed down to the subclass (R_{12}) in order for it to be accessible in *getPrice*.

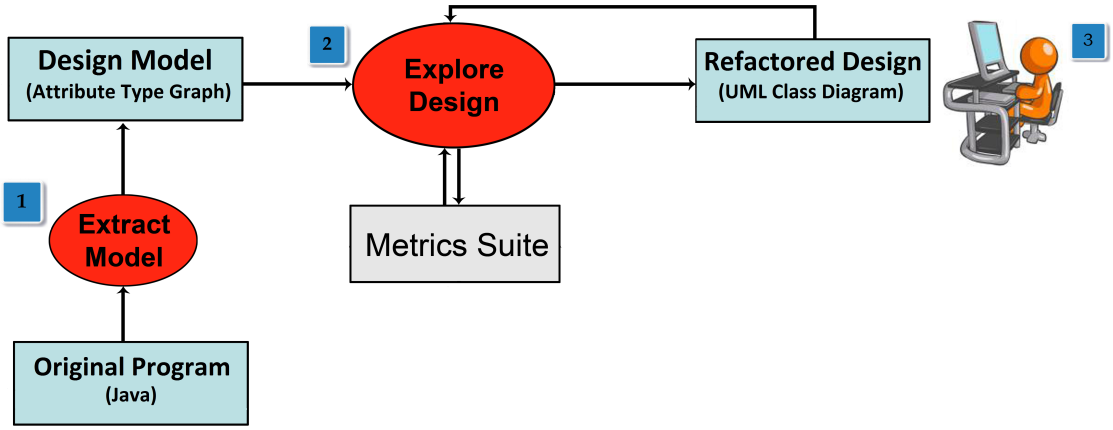


Figure 2: Typical workflow when using *Design-Imp*

- The above dependencies also reveal an implicit dependency between the two refactorings *Decrease Accessibility Method* (R_{14}) and *Decrease Accessibility Field* (R_{15}) with the *Extract Subclass* refactoring (R_2). Both *Decrease Accessibility* refactorings can be performed only after the class *Concert* has been created and the corresponding methods and fields have been moved to the newly created class.

As the example above demonstrates, there can be many relationships between refactorings, and even in this simple motivating example it is difficult to identify them all manually. As shown, the application of one refactoring may prevent certain other refactorings or make possible certain other refactorings. What makes the refactoring process more difficult is that the effect of each refactoring is only seen after the refactoring is applied to the design. The preconditions of a refactoring fail at its turn even though they were satisfied at the start of the sequence, and vice versa.

We describe our automated scheduling approach (REDaCT and REDaCT+) that addresses these problems in Sections 4 and 5, but we first present some preliminary information about the software tools we use in this work, and precise definitions of conflict and dependency.

3. Preliminaries

This section provides some necessary preliminary information about the software tool employed in our experiments (Section 3.1) and formal definitions for conflict and dependency in a refactoring sequence (Section 3.2).

3.1. Design-Imp

The investigations described in this paper make use of a software tool named *Design-Imp*. *Design-Imp* is an interactive refactoring framework developed by the authors to facilitate experimentation in improving the design of existing programs. It refactors the software system at a higher level of abstraction than its source code. Figure 2 depicts a typical workflow when using *Design-Imp*.

Design-Imp takes Java version 7 source code as input, extracts design information from the source code using the extract model process and expresses the extracted information as an attributed type graph [6]. This graph is then refactored using an interactive evolutionary search technique to improve the program according to a fitness function, expressed in terms of standard software quality metrics such as a combination of cohesion and coupling metrics. The output comprises the refactored graph, expressed as a UML class diagram, as well as detailed refactoring and

No.	Class-Level Refactorings	Description
1	Rename Class	Changes the name of a class to a new name, and updates its references.
2	Extract Hierarchy	Adds a new subclass to a non-leaf class C in an inheritance hierarchy.
3	Extract Subclass	Adds a new subclass to class C and moves the relevant features to it.
4	Extract Superclass	Adds a new super class to class C and moves the relevant features to it.
5	Collapse Hierarchy	Removes a non-leaf class from an inheritance hierarchy.
6	Inline Class	Moves all features of a class into another class and deletes it.
7	Extract Class	Creates a new class and moves the relevant features from the old class into the new one.
	Method-Level Refactorings	
8	Push Down Method	Moves a method from a class to those subclasses that require it.
9	Pull Up Method	Moves a method from some class(es) to the immediate superclass.
10	Rename Method	Changes the name of a method to a new one, and updates its references.
11	Decrease Method Accessibility	Decreases the accessibility of a method, i.e from protected to private.
12	Increase Method Accessibility	Increases the accessibility of a method, i.e from protected to public.
13	Move Method	Creates a new method with a similar body in the class it uses most. Either turns the old method into a simple delegation, or removes it.
	Field-Level Refactorings	
14	Push Down Field	Moves a field from a class to those subclasses that require it.
15	Pull Up Field	Moves a field from some class(es) to the immediate superclass.
16	Move Field	Moves a field from a class to another one which uses the field most.
17	Rename Field	Changes the name of a field to a new name, and updates its references.
18	Decrease Field Accessibility	Decreases the accessibility of a field, i.e from protected to private.
19	Increase Field Accessibility	Increases the accessibility of a field, i.e from protected to public.
20	Encapsulate Field	Creates getter and setter methods for the field and uses only those to access the field.

Table 1: A list of refactorings provided by *Design-Imp*

metrics information. As most of the program detail, especially method bodies, has been abstracted away, faster precondition checking and refactoring execution is possible. The result of the refactoring process is a desired design based on the employed fitness function and confirmed by the developer.

Design-Imp uses AGG API[2] as a graph transformation engine [26,27] to implement graph transformation rules. Each rule (i.e. refactoring) includes a pattern that is specified by two graphs, left and right hand side, and a morphism between them. Transformation rules may specify negative and positive application conditions as transformation preconditions. A negative application condition (NAC) specifies certain structures that are forbidden, while a positive application condition (PAC) expresses certain structures that are necessary to perform a transformation. Currently, *Design-Imp* supports 20 refactorings shown in Table 1. In the rest of this paper we use the term *refactoring* instead of *transformation rule* when referring to a transformation on the graph.

Design-Imp defines a meta-model, expressed as a type graph, based on the syntax of the Java language in order to specify how a Java program should be represented as a graph. It is not possible to define all necessary Java constraints as a type graph, e.g. cyclical inheritance is hard to prevent, so we added some general constraints similar to those defined by Mens [16] to our model.

Design-Imp is also capable of detecting conflicts and dependencies between refactorings through the use of a static analysis technique provided by AGG API called *critical pair analysis*. Critical pair analysis computes all the potential conflicts and dependencies between refactorings based on the notion of independence of graph transformations [6]. Using this technique, *Design-Imp* can

[2] http://user.cs.tu-berlin.de/~gragra/agg/

No.	Conflict	Description
1	Delete - Use	A refactoring *deletes* a graph object that is *used* by another refactoring.
2	Produce - Forbid	A refactoring *produces* a graph structure that is *forbidden* by another refactoring.
3	Change - Use	A refactoring *changes* an attribute value of a graph object in such a way that it can no longer be *used* by another refactoring.
No.	**Dependency**	**Description**
1	Produce - Use	A refactoring *produces* a graph object that is *used* by another refactoring.
2	Delete - Forbid	A refactoring *deletes* a graph objects that is *forbidden* by another refactoring.
3	Change - Use	A refactoring *changes* an attribute value of a graph object in such a way that it can be *used* by another refactoring.

Table 2: Relationships that can be detected between refactorings using AGG [23]

distinguish three kinds of conflict and three kinds of dependency between refactorings as described in Table 2. Definitions for conflict and dependency are presented next in Section 3.2.

3.2. Definitions of Conflict and Dependency between Refactorings

In this section we provide precise definitions for the concepts of *conflict* and *dependency*. These are concerned with the relationships between refactorings and are widely used in this paper.

Definition 1: *Dependency*
For two given refactorings (R_1 and R_2), R_2 is ***dependent*** on R_1 ($\boldsymbol{R_2 \rightarrow R_1}$) if R_2 can be applied after R_1, but not before that.

In this paper, as shown in Table 2, we distinguish three types of dependency between refactorings: *produce-use*, *delete-forbid*, and *change-use*. A *produce-use* dependency can happen if R_1 produces an element that is used by R_2. For example, in the motivating example, refactorings R_3 to R_{15} are dependent on one of R_1 or R_2. It is a kind of *produce-use* dependency as a method or field can be moved to a class only if the class has already been created.

As another example, consider a method that uses directly a *private* field in its own class. To push this method down to a subclass it is necessary first to increase the accessibility of the field to at least *protected* to make it accessible to the method in the subclass. In this case, the *Push Down Method* refactoring has a *change-use* dependency on the *Increase Field Accessibility* refactoring.

Definition 2: *Asymmetrical Conflict*
For two given refactorings (R_1 and R_2), R_1 has an ***asymmetrical conflict*** with R_2 ($\boldsymbol{R_1 \nrightarrow R_2}$) if R_2 cannot be applied after R_1.

In this paper, as shown in Table 2, we distinguish three kinds of conflicts: *delete-use*, *produce-forbid*, and *change-use*. As an example, a *delete-use* conflict between R_1, and R_2 can happen if R_1 deletes one or more elements (classes, methods, or fields) that are used by R_2.

Asymmetrical conflict is a one-way conflict. Thus, a conflict between R_1 and R_2 ($\boldsymbol{R_1 \nrightarrow R_2}$) does not imply that the application of R_2 will disable R_1. In addition, while an *asymmetrical conflict* is indeed a kind of *dependency*, we distinguish between them in this paper. In a conflict situation ($\boldsymbol{R_1 \nrightarrow R_2}$), both refactorings can be run individually, but R_2 cannot be run after R_1. However, in a dependency situation ($\boldsymbol{R_1 \rightarrow R_2}$), R_2 can only be run if R_1 is run first.

Definition 3: *Symmetrical Conflict*
For two given refactorings (R_1 and R_2), R_1 has a ***symmetrical conflict*** with R_2 ($\boldsymbol{R_1 \nleftrightarrow R_2}$) if

and only if they cannot both be performed on the design in any order, i.e. ($\boldsymbol{R_1} \nrightarrow \boldsymbol{R_2} \wedge \boldsymbol{R_2} \nrightarrow \boldsymbol{R_1} \Rightarrow \boldsymbol{R_1} \nleftrightarrow \boldsymbol{R_2}$)

As an example of a *symmetrical conflict*, consider a case where a method is moved from the same original class to two different target classes using two separate *Move Method* refactorings. While both refactorings are applicable, only one of them can be performed on the design. The other refactoring will fail subsequently as the method is no longer in its original class and so cannot be moved from there.

Definition 4: *Uninjurious Refactoring*
A refactoring with no *symmetrical* or *asymmetrical* conflict with any other refactoring is termed an *uninjurious* refactoring, in the terminology of Liu et al. [14]. This type of refactoring is of interest as it can be added to a refactoring sequence at any stage with no deleterious effect in terms of disabling other refactorings.

4. The REDaCT algorithm: Handling Conflict and Dependency in Software Refactoring Scheduling

In this section we describe one of the key contributions of this paper: the creation of a refactoring scheduling algorithm that can handle the conflicts and dependencies described in Section 3.2.

To find a valid refactoring sequence, we extend the conflict-aware scheduling approach proposed by Liu et al. [14]. They propose a heuristic algorithm to improve refactoring activities by arranging an application sequence for the available conflicting refactorings. Their approach computes *symmetrical* and *asymmetrical conflicts* between refactorings, where, in a conflict situation, the refactoring that has more effect on software quality, as defined by the QMOOD metric suite [1], has a higher priority than the other one. The solution we present here improves on the approach of Liu et al. in the following regards:

- The approach of Liu et al. only supports *delete-use* and *change-use* conflicts, and does not support *produce-forbid* conflicts, although they do propose this idea as future work. A *produce-forbid* conflict occurs when a refactoring produces an element or structure that is prohibited by the precondition of another refactoring [12]. Our approach handles all the conflict types in Table 2.

- The approach of Liu et al. does not support any kind of dependency between refactorings. In contrast, our approach is capable of detecting all inter-refactoring dependency types as shown in Table 2. By considering dependencies between refactorings, our scheduling algorithm is able to take into account the effect of a refactoring in terms of the other refactorings that it *enables*, whereas Liu et al. only consider cases where a refactoring *disables* other refactorings.

In Section 4.1 we describe how our refactoring scheduling algorithm, REDaCT, handles conflict and dependency relationships between refactorings. In Section 4.2 we discuss the strengths and weaknesses of this approach to refactoring scheduling.

4.1. The REDaCT Scheduling Algorithm

Our proposed scheduling algorithm, REDaCT is presented as pseudocode in Fig. 3. As illustrated, the algorithm takes as input the set of refactorings to be scheduled as well as a square matrix, called *RMatrix*, that contains information about how the input refactorings are related to each other. It is assumed that the refactorings are all beneficial, so a perfect solution is where all the refactorings

can be applied. REDaCT is a heuristic that attempts to find the longest possible valid sequence of refactorings that can be applied to the initial design.

RMatrix is computed by *Design-Imp* and contains information about conflicts and dependencies between refactorings. A character *'C'* in ($\mathrm{row_i}$, $\mathrm{column_j}$) of the matrix means an *asymmetrical conflict* exists between refactorings R_i and R_j, so applying R_i will prevent R_j from running. A *symmetrical conflict* will exist if ($\mathrm{row_j}$, $\mathrm{column_i}$) also contains a character *'C'*. On the other hand, a character *'D'* in ($\mathrm{row_i}$, $\mathrm{column_j}$) means that R_j is *dependent* on R_i, so R_j is only applicable if R_i has already been applied to the design. Note that a *symmetrical dependency* is an impossibility.

The REDaCT scheduling algorithm is depicted in Fig. 3. Five critical steps in the algorithm are highlighted and are elucidated in the paragraphs below:

Step 1: In the first step, it is necessary to find refactorings that are not dependent on any refactorings as well as having no conflict with other refactorings. A refactoring with no *symmetrical* or *asymmetrical* conflict with other refactorings is selected in order to prevent it from being disabled by other refactorings that might have an *asymmetrical conflict* with it [14]. However, such an uninjurious refactoring (see Section 3.2) is only selected if it is also not *dependent* on any refactorings except those already added to the refactoring sequence. This step guarantees that, where possible, all refactorings upon which a candidate refactoring is dependent are added to the refactoring sequence early in the process.

After a refactoring is added to the refactoring sequence, its corresponding row and column is removed from *RMatrix* as well. The algorithm may terminate at the end of first step if all refactorings have been added to the refactoring sequence. This only happens if there is no *symmetrical conflict* between any pair of refactorings in the set.

Step 2: In the second step, assuming that *RMatrix* is not empty, the score for each applicable refactoring R_c is computed using the following formula:

$$\mathrm{score(R_c) = directEffect(R_c) + positiveEffect(R_c) - negativeEffect(R_c)} \quad (1)$$

It is assumed that each refactoring in the refactoring set has a positive effect, i.e. that the developer has selected only refactorings that have a positive effect on the design of the program. Therefore, the maximum quality improvement is obtained if all refactorings in the refactoring set are applied to the initial design. Hence, we set the *directEffect* of each refactoring to 1, meaning that the application of each refactoring leads the refactoring process one step closer to the maximum achievable quality improvement. (Later, in Section 5, we will use a more sophisticated approach for computing the direct effect of a refactoring.)

The application of a refactoring R_c enables refactorings that are dependent on R_c to be run, assuming that they are not dependent on other available refactorings (see Section 3.2). In this paper, we count all these effects as the *positiveEffect* of the candidate refactoring R_c. So the positive effect of a candidate refactoring is the total number of refactorings that are enabled by it.

When a candidate refactoring R_c is applied to the design, it also disables other refactorings, R_o, with which it has an *asymmetrical conflict* [14]. In addition, if R_o is disabled, its dependent refactorings are disabled as well. In this paper, we count all these effects as the *negativeEffect* of the candidate refactoring, R_c. So the negative effect of a candidate refactoring is the total number of refactorings that are disabled by it.

Step 3: In the third step, the refactoring with the highest score is selected and added to the refactoring sequence. Since the score is based on the number of refactorings that will be disabled or enabled by the refactoring, the selection of a high-scoring refactoring promotes refactorings that increase the number of refactorings that can be selected in subsequent iterations.

Step 4: After the best refactoring is added to the refactoring sequence, it is necessary to update the scoring of refactorings that have been positively affected by the application of this refactoring. This includes refactorings that have an *asymmetrical conflict* with the selected refactoring [14], as well as refactorings that are *dependent* on the selected refactoring. The score for these positively affected refactorings is updated using Eq. 2 below. As shown, the merit of the selected refactoring is added to its affected ones in order to increase their chance of being selected in subsequent iterations.

$$\text{score}(\text{R}_{\text{affected}}) = \text{score}(\text{R}_{\text{affected}}) + \text{score}(\text{R}_{\text{selected}}) \tag{2}$$

As shown in Fig. 3, after the best refactoring is added to the refactoring sequence, all newly disabled refactorings are also removed from *RMatrix* to prevent them from being needlessly selected in subsequent iterations.

Step 5: In this step, refactorings that are neither dependent on, nor in conflict with, any remaining refactorings are added to the refactoring sequence. They can be safely applied at this stage, and doing so immediately prevents such a refactoring from being subsequently disabled by a refactoring with a better score that has an *asymmetrical conflict* with it.

At the end of the algorithm, *refactoringSeq* will contain the longest sequence of refactorings found that can be applied to the initial design.

4.2. Summary

To summarise this section, we have presented the REDaCT algorithm, which is our novel approach to refactoring scheduling that extends the state of the art [14] by handling a more extensive range of conflicts and dependencies. This algorithm is evaluated later in Section 6. REDaCT ignores the effect the refactorings have in terms of how close they bring the program to its desired design. In the next section we address this issue.

5. The REDaCT+ algorithm: Improving Refactoring Scheduling by Estimating the Contribution of Refactorings to Achieving the Desired Design

The approach proposed by Liu et al. uses the QMOOD metric suite [1] to measure the effect of refactorings on software quality. However, applying refactorings to the design and measuring their effect requires considerable effort. In addition, as no dependency between refactorings is detected by Liu et al., the impact of each refactoring is measured individually, and that cannot capture the real effect of a sequence of refactorings. In Section 5.1 below we describe a known, string-based approach to comparing software designs and put this to novel use in Section 5.2 to introduce a novel, lightweight approach to measuring the effect of a refactoring without actually applying the refactoring to the design. Finally, in Section 5.3, this approach to measuring refactoring effect is included in the scheduling algorithm to improve the accuracy of the scheduling approach; we term this extension *REDaCT+*.

5.1. Measuring Similarity between Software Designs

In this paper, an improvement in quality means an improvement in the similarity between the initial and desired designs. Therefore, during the refactoring process, a refactoring that improves the similarity between the initial and desired designs has priority over other refactorings.

Input: *refactoringSet:* set of refactorings to be scheduled.
Input: *RMatrix:* square matrix contains relationships between refactorings.
Output: *refactoringSeq:* contains a valid order of refactorings.

```
procedure SCHEDULING ALGORITHM(refactoringSet, RMatrix)
    refactoringSeq = null
    while (hasUninjuriousRefactoring()) do                          ▷ Step 1
        refactoringSeq.add(pickUninjuriousRefactoring())
        updateRMatrix()
    end while
    if (!RMatrix.isEmpty()) then
        measureScore()                                              ▷ Step 2
        repeat
            refactoringSeq.add(pickBestRefactoring()))              ▷ Step 3
            updateScores()                                          ▷ Step 4
            updateRMatrix()
            while (hasUninjuriousRefactoring()) do                  ▷ Step 5
                refactoringSeq.add(pickUninjuriousRefactoring())
                updateRMatrix()
            end while
        until (RMatrix.isEmpty())
    end if
    return refactoringSeq
end procedure
```

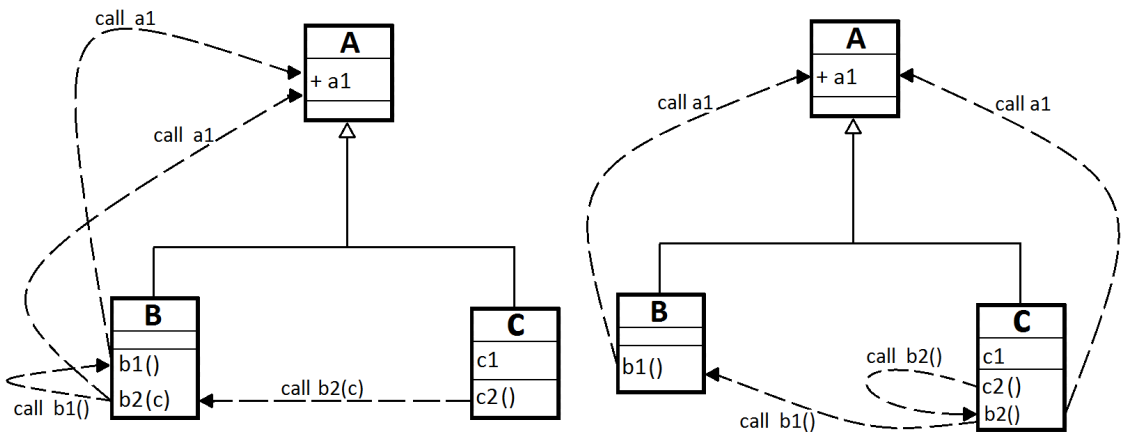
The functions used are defined as follows:
hasUninjuriousRefactoring(): Returns *true* if *RMatrix* contains at least one independent and uninjurious refactoring. Otherwise, returns *false*.
pickUninjuriousRefactoring(): Returns the first independent and uninjurious refactoring.
pickBestRefactoring(): Returns the refactoring with the highest score.
updateRMatrix(): The selected refactoring is removed from *RMatrix*(). The refactorings with which the selected refactoring has a conflict are removed from *RMatrix*() as well.
measureScore(): Computes the score for all remaining refactorings using equation 1.
updateScores(): Updates the score for the affected refactorings using equation 2.

Figure 3: The REDaCT algorithm. It orders the input refactorings to create the longest possible applicable refactoring sequence, in the presence of conflict and dependency between the refactorings.

To measure the degree of similarity between two designs, REDaCT+ uses a sequence alignment algorithm called *Fast Optimal Global Sequence Alignment Algorithm (FOGSAA)* developed by Chakraborty and Bandyopadhyay [3]. This algorithm is capable of finding the best alignment between two input strings with a lower computational complexity than other global alignment approaches. Full details of this algorithm are presented in the paper cited above.

To use the FOGSAA alignment algorithm in REDaCT+, the first step is to represent program's features such as classes, methods, fields etc. as a sequence of characters. In this paper, we use the approach proposed by Kessentini et al. [11] as a method to represent program elements as a string. Each element in the input Java program is represented using a specific character as follows: *Class (C), generalization relationship (G), realization relationship*[3] *(I), attribute (A), method (M), method parameter (P)*, and a coupling between two classes *(R)*. As an example, the representation of class *B* shown in Fig. 4a is *CGMMPR*. This sequence shows that the class inherits from another class, has a coupling relationship with one other class in the program and contains two methods, where the second method has a parameter.

[3] Realization in Java is the relationship between a class and an interface that it implements.



(a) The initial design: *CA (class A) CGMMPR (class B) CGAMR (class C)*

(b) The refactored design: *CA (class A) CGMR (class B) CGAMMRR (class C)*

Figure 4: How refactoring effect is expressed on the string representation of a design.

However, our representation differs from that of Kessentini et al. [11] in two significant ways. Firstly, in their work each character includes more detailed information such as name, type, accessibility etc. depending on the program element it represents. However, in our approach the element name is the only information that is included with each character. Secondly, in Kessentini et al. [11] every method invocation or field reference is represented by one R character. Therefore, if a class invokes a method in another class n times, n R characters are added to the resulting string. However, the number of accesses to fields and methods in a class is usually far greater than the number of fields and methods in the class, so this approach overemphasises the importance of R relationships over the other types when measuring similarity. To improve the efficiency of the alignment algorithm, we use a single R character to denote a coupling from the original class to another class without counting the number of connections between the two classes.

5.2. Expressing Refactoring Effect on the String Representation of a Program

Expressing the program as a sequence of characters and using an alignment algorithm to measure similarity between strings helps in measuring the effect of refactorings without actually applying them to the design. However, in order to do this it is necessary to determine first how the resulting string should be changed when a refactoring is applied to it.

Fig. 4 illustrates an example of how the *Move Method* refactoring changes the software design and the related string representation. In this example, method *b2(c)* is moved from its original class, named B, to a related target class named C. Figures 4a, and 4b show the UML design and the related string before and after refactoring respectively. Because the classes are related through the method parameter, after refactoring the input parameter is removed from the method signature. As illustrated, the sequence that shows class A (the first part in each sequence indicated by CA) is not changed as the refactoring has no effect on that. However, both sequences related to class B, and C (the second and third parts in each sequence) are changed because of the refactoring.

For each of the refactoring types in Table 1, its effect on the string representation of a program design is defined in a similar way as described for the *Move Method* refactoring. This enables us to estimate quickly the approximate effect of a refactoring without having to operate on source code parse trees. Note that the effect of a refactoring is only measurable when all refactorings upon which it depends have been applied to the design, e.g. the effect of a *Move Method* refactoring is

only measurable if the target class has already been created. We use a topological sort to create a linear ordering of the refactorings based on their dependency to ensure that all refactorings upon which a refactoring depends precede it in the ordering. Note that the refactoring sequence produced by the topological sort algorithm does not show the optimal ordering between refactorings as it does not take conflicts between refactorings into account.

The quality improvement achieved by a refactoring is thus measured by the FOGSAA string alignment algorithm. It expresses the difference in similarity between the current and desired designs before and after the applied refactoring. Using this approach, we can determine, for any refactoring under consideration, to what extent it contributes to achieving the desired design. In the next section we include this measure in the refactoring scheduling approach.

5.3. Including Refactoring Effect in the Scheduling Algorithm

The REDaCT scheduling algorithm described earlier in Section 4 tries to order the refactorings so that the maximum number of refactorings can be applied to the design. However, finding the longest sequence of refactorings is not always the best option to order refactorings, especially if there are *symmetrical conflicts* between refactorings, and different refactorings make different contributions to achieving the desired design.

To improve the scheduling algorithm, we extend it to include the contribution of the refactoring to achieving the desired design. We term this contribution the *refactoringEffect*, and it is measured as described in Section 5.2. Thus, the scoring function defined by Eq. 1 in Section 4 is changed as below. As shown, the default value for *directEffect* used in Eq. 1 is changed from 1 to the contribution of the refactoring on the similarity between designs:

$$\text{score}(\text{R}_\text{c}) = \text{refactoringEffect}(\text{R}_\text{c}) + \text{positiveEffect}(\text{R}_\text{c}) - \text{negativeEffect}(\text{R}_\text{c}) \quad (3)$$

All components of this summation are equally weighted. Thus the decision on whether to accept a refactoring depends equally on the contribution the refactoring makes to the desired design, the effect of refactorings it enables and the effect of refactorings it disables. In Section 6 this new approach, REDaCT+, is evaluated and it is compared with the vanilla REDaCT algorithm.

6. Evaluation

We have presented an algorithm for refactoring scheduling in the presence of conflict and dependency (REDaCT) and augmented this algorithm to exploit a desired design, if one is available (REDaCT+). In this section we evaluate these algorithms by applying them to a number of examples and assessing the results.

This section is structured as follows. In Section 6.1 we test the correctness of the REDaCT algorithm by applying it to the Event system described in the motivating example of Section 2. In Section 6.2 we demonstrate the necessity for the REDaCT+ algorithm, and evaluate this algorithm. In Section 6.3 we evaluate the ability of the REDaCT+ algorithm to schedule a 'noisy' refactoring sequence to achieve a desired design, while in Section 6.4 we evaluate the REDaCT+ algorithm on a medium-sized open source application. In Section 6.5 we summarise the results of our experiments.

6.1. Testing the correctness of the REDaCT algorithm

To test that the scheduling algorithm operates correctly, we applied it to the Event system that was used as a motivating example in Section 2. The aim is to determine if our refactoring scheduling

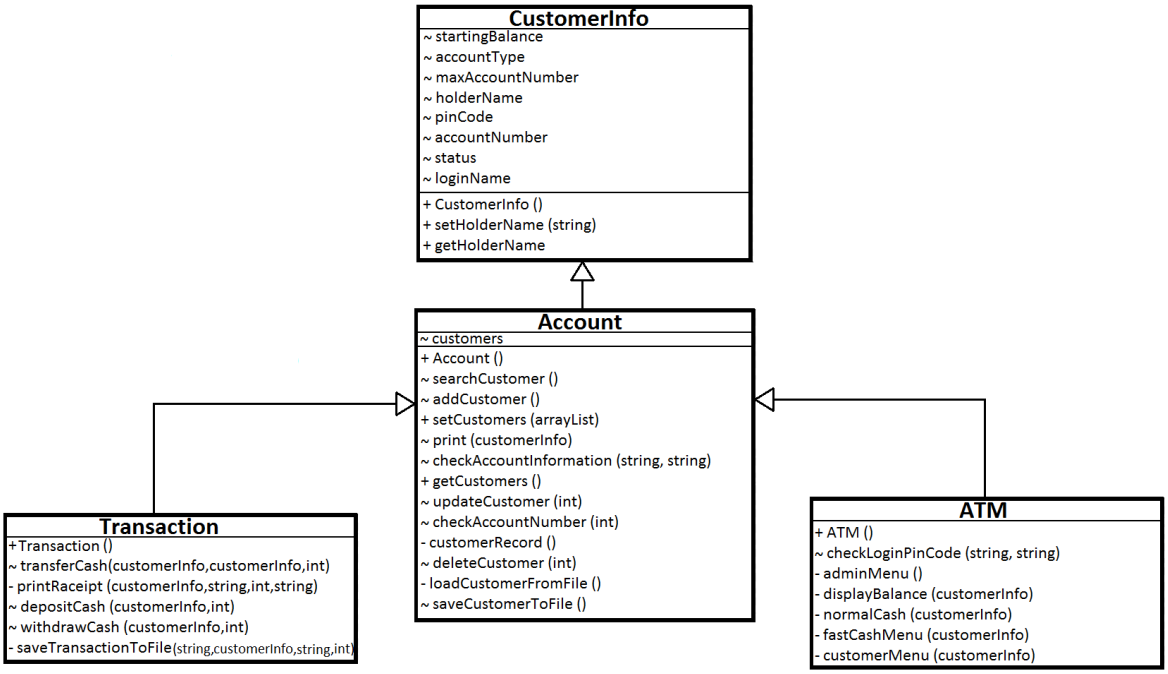


Figure 5: UML class diagram of an ATM application after refactoring

algorithm can order the refactorings in such a way as to generate the desired design shown in Fig. 1b from its initial one depicted in Fig. 1a. The refactorings in question are R_1 to R_{15} as defined in Section 2. As detailed in that section, a considerable amount of conflict and dependency exists between these refactorings and it is not immediately clear if they can all be applied to the initial design or not, so this forms a robust test for the REDaCT algorithm.

Applying REDaCT to the refactoring set shown in Section 2 yielded an ordering that enabled all 15 refactorings to be applied to the design as follows: R_1, R_3, R_5, R_8, R_4, R_2, R_6, R_7, R_{10}, R_9, R_{11}, R_{12}, R_{13}, R_{14}, R_{15}. The resulting design was identical to the refactored design, meaning that the refactorings were indeed performed in the correct order. As no *symmetrical conflict* was detected between the input refactorings, all 15 refactorings could be applied to the design.

The example used here is small, but the conflicts and dependencies between the refactorings are more complicated than would usually be encountered in a real-world system. In a larger system, typically only a few refactorings are applied to a class and its immediate relatives, so the level of conflict and dependency tends to be lower and sparser than in the example we use here. Nevertheless, when such conflicts and dependencies occur, they have to be addressed.

The result we obtain above demonstrates the ability of the REDaCT algorithm to handle the conflicts and dependencies between refactorings and hence to find an effective application order for the given refactorings.

6.2. Contrasting the REDaCT and REDaCT+ algorithms

The example we use here is a simplistic Automatic Teller Machine (ATM) simulation application [19]. It was developed by an inexperienced Java programmer, and so we expect that its design is not optimum and is easy to improve. Using *Design-Imp*, this ATM application was refactored using a fitness function defined as a combination of the two software metrics SCC (Similarity-based Class Cohesion) [4] and DCC (Direct Class Coupling) [1]. Table 3 depicts the refactoring sequence $R_1..R_{10}$ that led to the design for the ATM system depicted in Fig 5.

No.	Refactoring	Feature	Source class	Target class
R_1	Move Method	displayBalance()	Transaction	CustomerInfo
R_2	Move Method	displayBalance()	CustomerInfo	ATM
R_3	PullUp Method	printReceipt()	ATM	Account
R_4	Encapsulate Field	customers	Account	
R_5	PushDown Method	print()	CustomerInfo	Account
R_6	PullUp Method	printReceipt()	Account	CustomerInfo
R_7	Decrease Method Accessibility	displayBalance()	ATM	
R_8	Encapsulate Field	holderName	CustomerInfo	
R_9	Move Method	printReceipt()	CustomerInfo	Transaction
R_{10}	Decrease Method Accessibility	printReceipt()	Transaction	
R_{11}	Move Method	printReceipt()	ATM	Transaction
R_{12}	Move Method	displayBalance()	Transaction	ATM

Table 3: Sequence of refactorings applied to the ATM application. R_1 to R_{10} represent the sequence produced by *Design-Imp* in creating the design of Fig. 5. R_{11} and R_{12} were both added by hand to test the REDaCT+ algorithm.

When REDaCT was applied to the original ATM design and the set of refactorings, it produced the refactoring sequence R_8, R_5, R_1, R_2, R_4, R_3, R_6, R_9, R_7, R_{10}. This refactoring sequence is correct in that it yields the desired design when applied to the original ATM program design. However it is apparent that during the refactoring process the methods *printReceipt()* and *displayBalance()* are needlessly moved around various classes before being placed in their final target class.

To test if REDaCT is capable of finding a better sequence, we added two new *Move Method* refactorings, R_{11} and R_{12}, to the refactoring set. These new refactorings directly move the methods *printReceipt()* and *displayBalance()* from their original class to their correct target class. These refactorings are highlighted in grey in Table 3. The addition of the new refactorings created two *symmetrical conflicts* as follows: $R_1 \nleftrightarrow R_{12}$, and $R_3 \nleftrightarrow R_{11}$, and, because of the dependencies among the refactorings, three new *symmetrical conflicts* as follows: $R_2 \nleftrightarrow R_{12}$, $R_6 \nleftrightarrow R_{11}$ and $R_9 \nleftrightarrow R_{11}$.

In this new situation, an optimal scheduling algorithm should select both new *Move Method* refactorings instead of other refactorings (R_1, R_2, R_3, R_6 and R_9), as the newly added *Move Method* refactorings move the methods directly to their target class, which is the clearest and most comprehensible solution. However, the REDaCT algorithm still selects the same sequence as it did before, without including R_{11} and R_{12} in the sequence. This happens because the REDaCT algorithm favours refactorings that in turn increase the number of refactorings that can be selected in subsequent iterations of the algorithm.

We tested if the REDaCT+ algorithm could find the optimum refactoring sequence for the example described above, where the vanilla REDaCT algorithm failed, and found that the improved REDaCT+ algorithm did indeed find the equally good, but shorter, order among the 12 input refactorings shown in Table 3. REDaCT+ moved the two methods *printReceipt*, and *displayBalance* directly to their target class using the two added *Move Method* refactorings R_{11} and R_{12}, and rejected the now-superfluous refactorings R_1, R_2, R_3, R_6 and R_9. REDaCT+ succeeded as it does not simply apply the maximum number of refactorings as REDaCT does, but it uses its knowledge of the design desired to select refactorings that moved the program design towards this desired design, as detailed in Section 5.

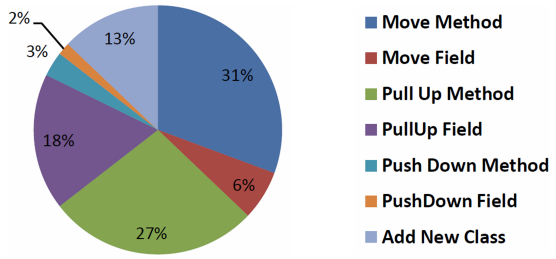


Figure 6: Breakdown of the 60 refactorings included in the refactoring set

6.3. Evaluating REDaCT+ on a 'noisy' refactoring sequence

To test how REDaCT+ would deal with a larger application, we applied it to the *Design-Imp* software itself. *Design-Imp* contains 65 classes that include 227 attributes and 600 methods, and so is larger than the earlier examples. In this experiment we also wanted to test how well REDaCT+ could deal with a 'noisy' refactoring set, one that contains redundant refactorings and so is a superset of the set of refactorings required to bring the initial program to its desired design.

The 'noisy' refactoring set was created as follows. First we use *Design-Imp* to automatically refactor the *Design-Imp* software twice in order to create two separate desired designs. As described in Section 3.1, the search-based algorithm used by *Design-Imp* is a stochastic one so each refactoring process yields a different set of refactorings but with the possibility of some commonality between the two refactoring sets. Fig 6 shows a breakdown of the combined set of refactorings produced by both refactoring processes. The 'noisy' refactoring set then is the union of these two sets. Duplicates were not removed, so this set (technically a multiset) contained 60 refactorings in total. In fact only one refactoring appeared in both refactoring sets, so the combined set contained 59 unique refactorings. A total of 3 conflicts and 298 dependencies were found to exist in this combined refactoring set.

We then selected one of these resulting designs as the final desired design and used REDaCT+ to try to refactor the initial design towards the selected final desired design. This is a robust challenge, where both the conflict and dependency aspects of the base REDaCT algorithm have to combine with the 'desired design' aspects of the REDaCT+ algorithm in order to filter out the unnecessary refactorings and attempt to produce a refactoring sequence that yields the given desired design.

REDaCT+ was able to find a refactoring sequence that was 93% correct and brought the program design to one close to the given desired design. In two cases the algorithm categorised two correct *Move Field* refactorings as detrimental even though both refactorings were critical to achieving the desired design. This problem occurred as *Design-Imp* incorrectly detected a spurious dependency between these two refactorings and another truly detrimental refactoring.

To sum up, this example shows the ability of the REDaCT+ algorithm to filter out refactorings that do not help in achieving the desired design. It also reveals how invalid inter-refactoring dependencies detected by *Design-Imp* can affect the accuracy of the REDaCT+ algorithm, leading to two correct refactorings being ignored. From this we see that the ability of REDaCT+ to correctly order a refactoring set is dependent on the accuracy of the detected conflicts and dependencies between the refactorings.

6.4. Evaluating REDaCT+ on an open source example

To investigate how REDaCT+ works with a non-trivial open source application, we used *JGraphX*[4] as an application to investigate. *JGraphX* is a medium-sized Java library that is used to display interactive diagrams and graphs and comprises 188 classes, 1356 attributes and 2908 methods.

In this experiment, we used *Design-Imp* to automatically refactor the design of the original *JGraphX* application to create a desired design to use in our experiments. We then used *Design Imp* to generate a refactoring set from the differences between the two designs, which yielded a refactoring set containing 50 refactorings.

To test the REDaCT+ algorithm, we applied it to the initial and desired designs of the *JGraphX* application as well as the generated refactoring set. The challenge here is that this is an open source application that the experimenters have no *a priori* knowledge of, and the refactoring set is non-trivial in size. However, REDaCT+ produced a refactoring sequence that transformed the initial design to the desired design with 100% success and with no spurious refactorings appearing in the sequence.

Note that in a large software system there are many possibilities for refactoring, and the size of the system tends to lead to less dependency and conflict between applied refactorings. However, as mentioned, the aim of this experiment was to show the proposed approach can be used for larger applications as well.

6.5. Evaluation Conclusion

The goal of this evaluation section was to provide an overall assessment of how the REDaCT and REDaCT+ algorithms perform when faced with a variety of challenges.

We demonstrated the basic correctness of the REDaCT algorithm in Section 6.1 by showing that it able to correctly sequence a heavily conflicted and interdependent set of refactorings. While the refactoring set was small in size, the intensity of the conflict and dependency was far greater than that found in our tests with an open source example in Section 6.4.

In Section 6.2 we demonstrated that, when several options exist, REDaCT fails to find the shortest refactoring sequence but that REDaCT+ is able to succeed by using its extra knowledge of the desired design that the refactoring sequence is trying to achieve. This established the case for the the REDaCT+ algorithm and this is the algorithm that we evaluate further.

In Section 6.3 we demonstrated that the REDaCT+ algorithm can schedule a 'noisy' refactoring sequence that has many optically-relevant but useless refactorings. A perfect solution was not achieved, but the 93% success rate is very satisfactory. In effect this means that a design very close to the desired design is achieved, and it is left to the developer to perform the final refactoring steps by hand.

In our final evaluation in Section 6.4 we demonstrated that the REDaCT+ algorithm can find a correct refactoring sequence when trying to schedule a non-trivial refactoring set (50 refactorings) on a medium-sized open source application. This demonstrates that REDaCT+ can perform well in a more realistic context.

We found that the size of program under investigation has only a minor effect on the speed of the REDaCT+ scheduling algorithm. The only time-consuming part is the algorithm used to identify conflicts and dependencies between refactorings. The number of comparisons to find conflict and dependency between refactorings is equal to $n * (n - 1)/2$, where n is the number of refactorings included in the refactoring set.

[4] http://www.jgraph.com/jgraphdownload.html

7. Related Work

The work related to this paper can be divided into three research areas: ranking refactoring opportunities, search-based refactoring and scheduling refactoring. These topics are discussed below.

Ranking refactoring opportunities involves sorting refactoring opportunities according to one or more criteria such as their impact on the overall design quality. Tsantalis and Chatzigeorgiou [30] propose a semi-automatic approach to identifying refactoring opportunities related to code smells based on system history and find that a refactoring opportunity involving a highly changeable code fragment is most likely to be improved through refactorings in the future, and therefore such refactorings should have a higher priority than others. In other work they propose an approach for detecting Move Method refactoring opportunities based on code smells [28] and also propose an approach to finding refactoring opportunities that introduce polymorphism as an alternative to state checking [29]. In these approaches, detected refactoring opportunities are ranked based on their impact on the overall design quality. However, the effect of refactorings is measured individually, without considering impact of refactorings on one another, and so it cannot guarantee to find the best sequence in which to apply the proposed refactorings.

In contrast to the aforementioned semi-automatic approaches, other research works aim to automate the whole process by using *search based refactoring* to find and apply a sequence of refactorings to a program [22]. In this approach, the process of refactoring is guided by a fitness function and a refactoring is accepted only if it improves the merit of the design based on metrics included in the fitness function. This approach has been used for several purposes: software quality improvement [5, 20], to fix code smells [11, 21] and to apply design patterns [10]. Although search-based refactoring techniques help to automatically find a close-to-optimal sequence of refactorings based on the employed fitness function, they do not usually generate the most effective refactoring order.

In the remainder of this section we examine closely related research that also aims to detect *dependencies* and *conflicts* between refactorings in order to sort refactorings into an optimum order for application. Mens et al. [17] use parallel and sequential dependency analysis to detect relationships among a set of input refactorings. Like our approach, the program and refactoring activities are considered as a graph and graph-based rules respectively. However, they only focus on specifying relationship between refactorings without introducing a practical algorithm on how the refactorings should be arranged. Further, in their work, sequential dependencies are only detected after a candidate refactoring is applied to the design and then find out which refactorings become applicable or inapplicable. Thus, the approach cannot detect existing sequential dependencies between refactorings at once without applying refactorings to the source code.

Zibran and Roy [33] introduce a scheduling approach based on three factors: maximised quality gain measured based on standard software quality metrics, minimised refactoring effort estimated by a proposed refactoring effort model and satisfaction of higher priorities defined manually by the developer. They formulate the scheduling of code clone refactorings as a *constraint satisfaction optimisation problem*, and use *constraint programming* to implement the proposed model.

Estimation of clone refactoring effort using an effort model has also been investigated by Bouktif et al. [2]. They use a simple genetic algorithm to schedule clone refactoring activities in order to achieve the greatest quality improvement with minimum resource consumption while satisfying priority constraints. However, they ignore conflicts between clone refactorings and assume that duplicated codes can be refactored independently, which is not a correct assumption.

Among the research works focused on scheduling code smell refactoring, the work of Liu et al. [14] is most closely related to ours. They propose a heuristic algorithm to schedule code smell refactorings. We extend their refactoring scheduling algorithm by considering not just *conflicts* between refactorings but also *dependencies*. Furthermore, we take into account a type of refactoring

conflict not handled in their work, where the application of one refactoring violates the precondition of another. We also introduce the idea of a desired design and show how it can be included in the scheduling algorithm to guide the refactoring process more effectively. We measure not only the effect of a refactoring in terms of its direct contribution to achieving the desired design, but also its indirect contribution in terms of the refactorings it enables and disables. Later work by Liu at al. [15] looks at the related problem of scheduling code smell detection and resolution when a software system is radically refactored in 'batch' mode. They show that the order in which code smells are addressed is significant due to the overlap between smells, and show that refactoring effort can be reduced significantly (by up to 20%) by appropriate scheduling.

Lee et al. [13] also take into account that a refactoring can also *enable* other refactorings. They use a genetic algorithm to identify an appropriate refactoring schedule for code clones. To support both cases, the original refactoring set is updated according to changes applied in the program after the refactoring is performed. The effect of each refactoring sequence expressed as a chromosome is measured using the QMOOD quality model [1]. However, the fitness evaluation is expensive due to the fact that each chromosome must be individually applied to the system and then its effect on quality measured. It is different from our approach, where refactorings are applied to a sequence of characters (representing the source code) and each refactoring is simulated only once and its effect on the quality is measured at that time.

8. Conclusions and Future Work

In this paper we presented the REDaCT algorithm, our approach to refactoring scheduling in the presence of inter-refactoring conflicts and dependencies that extends the state of the art [14] by handling a more extensive range of conflicts and dependencies. We also developed an extension to this algorithm, REDaCT+, that also takes into account the contribution of each refactoring towards achieving a given *desired design* for the software. To validate our proposed scheduling approach, we carried out evaluations on four examples: two small constructed examples, a software tool developed by the authors and a medium-sized open source software system. The results obtained demonstrated that REDaCT can order a refactoring sequence in the presence of conflicts and dependencies, that REDaCT+ can outperform REDaCT and that REDaCT+ works well even when many irrelevant refactorings are included in the refactoring set.

In future work we plan to explore further the process of creating the desired design, using an interactive process similar to that proposed by Simons et al. [24, 25]. This paves the way for the REDaCT+ refactoring algorithm to form part of a larger, interactive framework that helps the developer to create a desired design, and then refactors the code to comply with this design, a notion described in our earlier work [9]. In this context, more extensive evaluation with larger software systems and with software developers will be necessary.

Acknowledgement

This work was funded by the Irish Programme for Research in Third-Level Institutions (PRTLI) and in part by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Research Centre (www.lero.ie).

References

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

[2] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1885–1892, Seattle, Washington, USA, 8-12 July 2006. ACM.

[3] A. Chakraborty and S. Bandyopadhyay. FOGSAA: Fast optimal global sequence alignment algorithm. *Scientific reports*, 3, 2013.

[4] J. A. Dallal and L. C. Briand. An object-oriented high-level design-based class cohesion metric. *Journal of Information & Software Technology*, 52(12):1346–1361, 2010.

[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*, GECCO '07, pages 1106–1113, New York, NY, USA, 2007. ACM.

[6] R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 161–176. Springer, 2002.

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

[8] 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*, WRT '11, pages 41–44, New York, NY, USA, 2011. ACM.

[9] I. Hemati Moghadam and M. Ó Cinnéide. Automated refactoring using design differencing. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 43–52, Washington, DC, USA, 2012. IEEE Computer Society.

[10] 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*, pages 1341–1348, 2010.

[11] M. Kessentini, R. Mahaouachi, and K. Ghedira. What you like in design use to correct bad-smells. *Software Quality Journal*, 21(4):551–571, 2013.

[12] L. Lambers, H. Ehrig, and F. Orejas. Conflict detection for graph transformation with negative application conditions. In *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2006.

[13] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Software: Practice and Experience*, 41(5):521–550, 2011.

[14] H. Liu, G. Li, Z. Y. Ma, and W. Z. Shao. Conflict-aware schedule of software refactorings. *IET Software*, 2(5):446–460, 2008.

[15] H. Liu, Z. Ma, W. Shao, and Z. Niu. Schedule of bad smell detection and resolution: A new way to save effort. *IEEE Transactions on Software Engineering*, 38(1):220–235, 2012.

[16] T. Mens. On the use of graph transformations for model refactoring. In *Generative and transformational techniques in software engineering*, pages 219–257. Springer, 2006.

[17] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[18] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, 2008.

[19] O. Mursleen. Java code for atm operations, June 2010. http://freesourcecode.net/javaprojects/13191/Java-code-for-ATM-Operations#.VL_PC6YqYaA.

[20] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[21] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, 2013.

[22] O. Raiha. A survey on search-based software design. *Computer Science Review*, 4(4):203 – 249, 2010.

[23] O. Runge. The AGG 1.5.0 development environment - the user manual, 2014. http://user.cs.tu-berlin.

de/~gragra/agg/AGG-ShortManual/AGG-ShortManual.html.

[24] C. L. Simons and I. C. Parmee. Elegant object-oriented software design via interactive, evolutionary computation. *IEEE Transactions on Systems, Man, and Cybernetics: Part C Applications and Reviews.*, 42(6):1798–1805, November 2012.

[25] C. L. Simons, I. C. Parmee, and R. Gwynllyw. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Transactions on Software Engineering*, 36:798–816, 2010.

[26] G. Taentzer. AGG: A tool environment for algebraic graph transformation. In *Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *Lecture Notes in Computer Science*, pages 481–488. Springer, 2000.

[27] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2004.

[28] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35:347–367, 2009.

[29] N. Tsantalis and A. Chatzigeorgiou. Identification of refactoring opportunities introducing polymorphism. *Journal of Systems and Software*, 83:391–404, March 2010.

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

[31] Z. Xing and E. Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 263–274, Washington, DC, USA, 2006. IEEE Computer Society.

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

[33] M. F. Zibran and C. K. Roy. Conflict-aware optimal scheduling of prioritised code clone refactoring. *IET Software*, 7(3):167–186, 2013.