Models and Algorithms for Refactoring Statements

THESIS

Submitted to the Faculty of Science of the University of Fribourg (Switzerland) in conformity with the requirements for the degree of
Doctor scientiarum informaticarum

Submitted by

Nicolas JUILLERAT

of

Switzerland

Thesis No. 1646
Printed by UniPrint, University of Fribourg
2009
Accepted by the Faculty of Science of the University of Fribourg (Switzerland) following the proposal of:

- Prof. Ulrich ULTES-NITSCHE, University of Fribourg, Switzerland, President of the committee;
- Prof. Béat HIRSBRUNNER, University of Fribourg, Switzerland, Thesis Director;
- Dr. Michèle COURANT, University of Fribourg, Switzerland, Examinator;
- Prof. Oscar NIERSTRASZ, University of Berne, Switzerland, Examinator.

Fribourg, October 22, 2009

The Thesis Director:  The Dean:

Prof. Béat HIRSBRUNNER  Prof. Titus JENNY
Abstract

Refactoring is the process of adapting the design of existing code in order to cope with late changes of the requirements. This is realized by applying carefully chosen semantics preserving transformations on the existing code. This thesis presents new models and algorithms for the implementation of automated and semi-automated transformations of Java code used in complex refactorings.

The relevance of refactoring existing code is recognized. Various tools and techniques have already been developed to automate several of the corresponding transformations. We give an overview of the existing models, analyses and transformation algorithms, and discuss their strengths and weaknesses.

Then we explore an area that is still in an early stage of development: transformations of executable statements such as method bodies, as opposed to transformations of the declarative part of an application (methods, fields, classes and their relations). Coping with executable statements is necessary for instance to implement complex refactorings such as method extraction. We present a difficult refactoring as a case study, forming a template method, and show that it can hardly be implemented with state of the art methodologies. We discuss the underlying challenges, and propose appropriate novel solutions in the form of new models and algorithms, or extensions of existing techniques. Among the discussed topics, we tackle problems like effective code differentiation, fast data and control flow analysis based on the abstract syntax tree, and the automatic resolution of preconditions in method extraction.

Keywords: Refactoring, Source Code, Java, Form template method, Abstract Syntax Tree, Data flow, Control flow, Analysis, Transformation, Semantics preserving, Executable statements, Class Graph, Duplicated code, Clone detection, Code differentiation, Boolean flags, Extract method, LZ77, Longest Common Subsequence, Preconditions, Web frameworks.
Résumé

La refactorisation est un processus qui consiste à adapter l’architecture d’un programme existant afin de faire face à l’introduction tardive de changements et de nouvelles fonctionnalités. Concrètement il s’agit d’appliquer une ou plusieurs transformations sur le code source existant tout en préservant la sémantique. Cette thèse présente de nouveaux modèles et algorithmes permettant d’implémenter de manière automatique ou semi automatique les transformations de code nécessaire à des refactorisations complexes.

La pertinence de la refactorisation est reconnue dans le monde du logiciel, et de nombreux outils ont déjà été mis au point pour automatiser plusieurs transformations sous-jacentes. Nous donnons un aperçu des modèles existants, des algorithmes d’analyse et de transformation de code source, et nous discutons leurs forces et faiblesses.

Par la suite, nous explorons un domaine particulier de la refactorisation qui est encore immature: la transformation de code exécutable tel que le corps d’une méthode, par opposition aux transformations de la partie déclarative d’un programme tel que les méthodes, les champs, les classes et leurs relations. Il est en effet nécessaire d’être en mesure de manipuler du code exécutable par exemple pour implémenter de manière automatique une refactorisation comme l’extraction d’une méthode. Nous présentons comme étude de cas une transformation difficile: la création d’une méthode généralisée (“form template method”). Nous montrons que l’état de l’art ne permet pas l’implémentation de cette transformation. Nous discutons des problèmes sous-jacents, et proposons des solutions sous la forme de nouveaux modèles, de nouveaux algorithmes, ainsi que d’extensions de techniques existantes. Parmi les sujets discutés, nous abordons les problèmes suivants: la différenciation efficace de code source, l’analyse rapide des flux de données à l’aide d’arbres à syntaxe abstraite, et la résolution automatique de conditions préalables à l’extraction de méthodes.
# Contents

1 **Introduction** 4  
1.1 Why Refactor? 4  
1.1.1 A Concrete Example 5  
1.1.2 Under engineering 7  
1.1.3 Over engineering 8  
1.2 Refactoring Definition 10  
1.2.1 Definition of the Process 11  
1.2.2 Definition of the Context 12  
1.2.3 Definition of our Focus 13  
1.3 Automated Tools 13  
1.4 Highlight of our Contributions 14  
1.5 Structure of the Document 15  

2 **State of the Art** 17  
2.1 Modeling Source Code 17  
2.1.1 The Class Graph 18  
2.1.2 The Abstract Syntax Tree 21  
2.1.3 The Control Flow Graph 23  
2.1.4 The Data Flow Graph 25  
2.1.5 The Program Dependence Graph 26  
2.1.6 FOOD: a Data and Control Flow Model 27  
2.1.7 Other Models 30  
2.2 Analysis and Transformation Methodologies 30  
2.2.1 Graph Transformations 30  
2.2.2 Flow Analysis 33  
2.2.3 Clone Detection 35  
2.2.4 AST Transformations 39  
2.2.5 Domain Specific Languages 41  
2.2.6 Dataflow Languages 42  
2.2.7 User Interface 44  
2.3 Summary 45  

3 **Case Study: Forming a Template Method** 46  
3.1 Problem Description 46  
3.2 Challenges 54  
3.2.1 Code Differentiation 54  
3.2.2 How to Deal With the “Extract Method” Preconditions 57
Chapter 1

Introduction

This thesis is about models and algorithms for refactoring. To introduce the topic, Section 1.1 first motivates the need for refactoring by exploring the problems it has to solve. Then Section 1.2 defines refactoring, and presents the context of this thesis. Section 1.3 introduces the goal of the thesis, namely the implementation of automated and semi-automated tools for refactoring of Java code.

Section 1.4 briefly highlights the contributions that are brought by this thesis, and Section 1.5 gives the structure of the remainder of this document.

1.1 Why Refactor?

Refactoring became popular with the advent of the so called Agile Programming methodology [36, 55, 72], including the popular Extreme Programming (XP) [80]. Agile programming breaks many of the initial assumptions made since the birth of the programming discipline. Based on experience in the real-world industry, it takes the following facts as granted:

- It is impossible to know in advance all the features that will be requested;
- The design of an application is constantly changing while it is being developed;
- There is no clear distinction between design time and run-time. Design continues during run-time;
- Even the best initial design will degrade to chaos as new functionalities are added, unless efforts are given to adapt it;
- Technology will change more quickly than it is possible to upgrade existing legacy code

Agile programming recognizes through these facts a lack of stability in software due to constant changes. It proposes tools and methodologies to cope with change rather than trying to defend against it.

One of these methodologies is refactoring. Its purpose is to cope with late design changes and new requirements. It provides a systematic way of efficiently changing an existing design. “Existing design” implies that the underlying code has already been written. Hence the challenge in changing an existing design is to also appropriately modify the underlying code, and to do it without introducing bugs or changes to the existing semantics. Martin Fowler gives the following definition of refactoring [27, 28]:

“Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior”
In this thesis we are interested in automated and semi-automated algorithms to modify existing code. We do not discuss the various processes that precede this step, such as designing, writing code, or identifying the necessary design changes. Hence the code to modify and the actual modification are just considered as the inputs to the discussed algorithms.

1.1.1 A Concrete Example

In this section we describe how refactoring can help in the development process using a concrete example. This example is taken from a real-world scenario that occurred in a company where the author has been working.

Assume that you are requested to make an application to create, take and correct exams. Assume that an exam only consists of multi-choice questions, whose answers are given by checking one or more check boxes. Furthermore, exams and questions can be loaded and saved in a relational database.

To implement such an application, questions can easily be represented by a single `Question` class. Now imagine that your application has been finished, tested and successfully used for a while. Then you are asked to add two new features. First, the exam should now also consist of other question types. You are asked to add “text” questions, in which the answer is given by typing one or more words; and single-choice questions, in which the answer is given by selecting the correct entry in a combo box. Second, it should also be possible to load and save exams and questions into XML files.

At this place, it is clear that the single `Question` class is no longer the best design for the new version of the application. With the help of refactoring, it is possible to develop a new version of the application on the top of the old one by proceeding in two steps.

In a first step, the design of the current application is modified to suit the new requests; but no new functionalities are added at this time: the modified application should behave exactly as the old version. Figure 1.1 shows a possible change of design. The single `Question` class has been split into an abstract `AbstractQuestion` class and a concrete `MultiChoiceQuestion` subclass. Furthermore, a strategy pattern with a single concrete strategy is introduced for the loading and saving logic. The `load` and `save` method implementations are moved to the `RelationalStorage` class.

In a second step only, the new features are actually implemented. This basically consists in creating two new subclasses of the `AbstractQuestion` class, and one subclass of the `StorageStrategy` class. The changes performed in the second step are illustrated in Figure 1.2.

The first step is the actual refactoring process: the design of the application has been changed, but no functionality is modified or added. In other words, a semantics-preserving transformation of the existing
code has been applied. The second step is just regular engineering and coding (and will hence not be addressed in this thesis).

In practice, the two steps are not always clearly separated. Many authors have advocated to clearly separate them though [27, 47, 58, 68]. The main reason is that it allows one to completely test the application after the first step in order to verify that its behavior has not changed and that no new bugs have been introduced. This is especially practical if automated test cases are already present.

Looking closely at the refactoring example, we see that the process involves various small transformations:

- The class `Question` is renamed into `AbstractQuestion`.
- After the `MultiChoiceQuestion` subclass is created, methods and fields that are specific to multiple-choice questions are pushed down from the `AbstractQuestion` to the `MultiChoiceQuestion` class.
- After the strategy pattern is created, the `load` and `save` methods are moved to the `RelationalStorage` class.

All these small transformations are also refactorings. Indeed, many refactorings can be expressed by the composition of smaller refactorings, and can be combined with other refactorings into bigger ones. While books are presenting catalogs of small refactorings [27, 47], it should be noted that there is no definite list of possible refactorings. Basically, any transformation of the source code that results in source code and preserves the semantics can be considered as a refactoring.

In practice, the goal of the refactoring process is not just to make arbitrary and random semantics preserving changes of the code, but to do changes that actually improve the existing design. There can be several reasons for which a design needs to be improved:

- New features are requested (as in the previous example). It is rare that an initial design can directly accommodate all the possible new features. While this can be a topic for debate, Agile refactoring methodologies (from which refactoring originates) take this fact as granted.
- Bad design. Even if no feature needs to be added or modified, the initial design may just have been poor or inadequate right from the start. Refactoring provides tools to improve it, and hence to enhance the clarity of existing code.
• Removal of features. Removing several features for instance can result in class hierarchies or design patterns that are no longer relevant. In that case, refactoring can be used to simplify an existing design.

Several small refactorings (such as renaming or moving fields or methods) are well defined enough so that they can be semi-automated in development environments. As bigger refactorings are mostly combinations of smaller refactorings, the automation of small refactorings is also relevant for bigger ones.

1.1.2 Under engineering

In the previous example, we showed how an existing design can be modified to better accommodate new features. But what would happen if we did not do any refactoring? Indeed a mistake that is frequently seen in practice is to renounce to any modification of the design and to stick to the current one when requests for new features are made. Doing so with our example would result to the following kind of code:

```java
public void save(int where, ...) {
    if (where == RELATIONAL_DB) {
        switch (this.questionType) {
            case QuestionType.MULTI_CHOICE:
                [Code to save a multi-choice question in a relational database]
                break;
            case QuestionType.SINGLE_CHOICE:
                [Code to save a single-choice question in a relational database]
                break;
            case QuestionType.TEXT:
                [Code to save a text question in a relational database]
                break;
        }
    } else if (mode == XML_FILE) {
        switch (this.questionType) {
            case QuestionType.MULTI_CHOICE:
                [Code to save a multi-choice question in XML]
                break;
            case QuestionType.SINGLE_CHOICE:
                [Code to save a single-choice question in XML]
                break;
            case QuestionType.TEXT:
                [Code to save a text question in XML]
                break;
        }
    }
}
```

Such code is unfortunately frequently seen in practice [36, 37]. In our simple example, it seems unlikely that a developer would stick to the single `Question` class. In practice though, the application and its design might be much bigger, and changing the design might seem as too much, or useless work. Furthermore, requests for new features rarely come all at once. Sticking to a given design might seem a
good choice if only a small change is requested. But as many small changes are requested one after the other, the result is then the same at the end.

In our example, imagine for instance that only single-choice questions are requested, and no other new features. As single-choice questions are very similar to multi-choice questions, it is quite appealing to keep the single Question class, because only a few changes are involved. Also imagine that it is only requested that exams could be saved in two different relational databases. In this case as well, only minor changes are involved, and it seems that the initial design does not need to be changed.

But as many small changes are requested, small and sparse conditional code gets increasingly more frequent in the code. After many small change requests, the code might be similar to our previous example anyway, with a high density of conditional code.

The problem with the above code is that such code is usually spread in the entire application. Adding a new question type in such code for instance, would require modification in the entire application. Modularity is lost. Furthermore, refactoring the above code is still possible in order to improve it, but is expected to be much more difficult than if it were done before the new functionalities are implemented.

Based on this observation, the extreme programming methodology includes as one of its rules to refactor mercilessly [80]. In other words, refactoring must be done as soon as possible, that is, as soon as the existing design starts getting slightly suboptimal.

1.1.3 Over engineering

Coming back to our initial example, another possibility that comes to mind is the following. Why not create a “complete” design right from the start; that is, a design that already includes a class hierarchy for different question types and strategies for loading and saving? There are unfortunately many problems with this approach, which is known as “over engineering”:

- It is impossible to know in advance what new features will be asked for. With our previous example, which was a real project in a company, we had another customer that asked for a similar application. Yet he never asked for different question types (only multi-choice was necessary) or different storage means. On the other hand, he asked for plenty of other features, such as a weighting scheme for the questions, and the possibility to ask for different kind of hints while taking an exam. A design such as the one in Figure 1.2 helps in no way for implementing these new features.

- A design will usually change while it is being used, as new features and modifications are requested. Furthermore, technology itself is expected to evolve as well, making even the best initial design obsolete.

- Even if we are able to build a perfect design that is ready for any kind of modification, it is likely that this design would be much more complicated than the one of Figure 1.2. As a result, the application will take much more time to be released and will also cost much more than it should. In practice though, it is more likely that the deadline will not be met and that the project will be abandoned.

Agile programming considers both under engineering and over engineering as mistakes. Refactoring can be used to fix both under and over engineering.

A serious problem with over engineering is that it is more difficult to identify than under engineering. Most of the time, over engineered code just looks elegant and well structured. Furthermore, the usually huge amount of design patterns in such code can make its author proud of it and hence reluctant to consider any changes [6, 37]. The code may look complex as well, but it is hard to say whether it is
unnecessarily complex (and hence over engineered), or if it is just modeling a complex problem. Here is a good illustration:

```java
public interface MessageStrategy {
    public void sendMessage();
}

public abstract class AbstractStrategyFactory {
    public abstract MessageStrategy createStrategy(MessageBody mb);
}

public class MessageBody {
    Object payload;
    public Object getPayload() {
        return payload;
    }
    public void configure(Object obj) {
        payload = obj;
    }
    public void send(MessageStrategy ms) {
        ms.sendMessage();
    }
}

public class DefaultFactory extends AbstractStrategyFactory {
    static DefaultFactory instance;
    private DefaultFactory() {
    }
    public static AbstractStrategyFactory getInstance() {
        if (instance == null) {
            instance = new DefaultFactory();
        }
        return instance;
    }

    public MessageStrategy createStrategy(final MessageBody mb) {
        return new MessageStrategy() {
            MessageBody body = mb;
            public void sendMessage() {
                Object obj = body.getPayload();
                System.out.println((String)obj);
            }
        };
    }
}

public class OverEngineering {
    public static void main(String[] args) {
        MessageBody mb = new MessageBody();
```

---

1Taken from [http://developers.slashdot.org/comments.pl?sid=33602&cid=3636102](http://developers.slashdot.org/comments.pl?sid=33602&cid=3636102). Another good example is [http://www.phppatterns.com/docs/design/hello_world_in_patterns](http://www.phppatterns.com/docs/design/hello_world_in_patterns).
mb.configure("Hello, world!");
AbstractStrategyFactory asf = DefaultFactory.getInstance();
MessageStrategy strategy = asf.createStrategy(mb);
mb.send(strategy);
}
}

This code looks elegant, flexible and well structured. It uses several design patterns, which seems to imply it has been carefully designed. Furthermore, every class and method is small and does little, so it is difficult to figure out anything that could potentially be simplified. But are you able to figure out what this code actually does? Here is the answer. The following code is semantically equivalent, but without any over engineering in it:

```java
public class RegularEngineering {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Over engineering is hard to define or even measure in practice. However there are common symptoms [36]:

- Implementing a simple change requires a lot of time, not because there are a lot of actual changes to do in the code, but because a lot of time is required for the programmer to understand the existing code, and to figure out which part of the code has to be modified.
- Implementing a simple change that was not “predicted” by the current design requires changes in several classes, if not a change of the whole architecture.
- The percentage of “business” code is very small. The “business” code denotes the code that actually performs what the application is supposed to do. In both previous examples, the application is supposed to display the “Hello, world!” message, hence only the “System.out.println(...)” statement belongs to the “business” code. If the rest of the code is proportionally too large, the chances of being in the presence of over engineering are high.

A common cause of under engineering is the failure to use a design pattern when it is the most appropriate solution to a problem. Reciprocally, a common cause of over engineering is the overuse of design patterns on situations in which they are not necessary (or not adequate), or on which straightforward language constructs (inheritance, events, generics, etc) are more appropriate.

A key notion of refactoring is that it can be used to either increase or decrease the complexity of a design. In other words, refactoring aims in solving both under engineering and over engineering problems. The choice of which direction to use depends on the actual code and the requirements. In practice, under and over engineering mainly occur as a result of requirement changes (although bad programmers are another possible cause), and are hence difficult to avoid unless refactoring efforts are investigated.

Note however that good programming practices also help in coping with several easy to predict changes [75]. Hence the use of refactoring is not a replacement of the initial design process but a complement.

### 1.2 Refactoring Definition

It is now time to define the notion of refactoring. We define it in terms of the process (what it has do do?), in term of the context (when it is used?), and in term of our focus in this thesis (what kinds of refactoring are we interested in?).
1.2.1 Definition of the Process

The General Definition

The previous section has shown the potential uses of refactoring on concrete situations. The general definition of refactoring is the process of improving the design of existing code. This process implies several different tasks [27, 47], such as:

- Identify bad designs, or designs that require changes to accommodate a new feature
- Identify the refactoring(s) to apply
- Choose the parameters of the refactoring (new name for a rename, method name for an extract method, etc).
- Apply the refactoring

Our Focus

In this thesis, we only focus on the last point, applying a refactoring, which is actually the only part that actually transforms the code. We also use the term “refactoring” throughout this thesis to denote only this particular process.

We assume that all the other tasks are already done. The results of these tasks are thus only considered as inputs of the methodologies that we develop. Whether these inputs are provided by the user, by an automated tool, or by a combination of them does not matter. With these assumptions, the definition of refactoring that we use is basically to implement a given semantics preserving transformation of the code correctly. While this is only a subset of the common definition of refactoring, this is the task that is discussed in the remainder of this thesis.

With this definition, a refactoring can also be viewed as a process that takes as input source code and some parameters (dependent on the desired transformation), and produces semantically equivalent source code as output.

Semantics Equivalence

A notion deserves some attention: “semantically equivalent”. This notion seems to be easy to define, and would mean that the transformed code is equivalent to the original one, meaning that it produces the same outputs given the same inputs. But there are some pitfalls to consider:

- Are two different codes that produce the same output semantically equivalent, even if they execute at different speeds? This might be a critical issue for instance in the context of real-time applications.
- What about multi-threaded applications? Within a single thread, two write accesses to two different variables that do not depend on each other can be swapped without changing the semantics. Yet if another thread reads these variables at the same time, the order in which they are written can have an influence on that other thread, and this may affect the semantics of the application.

Therefore, we restrict ourselves to the notion of semantic equivalence that has been adopted, for instance, by the just-in-time compiler of the Java virtual machine:

- Differences in execution speed are not considered as semantic differences. Note that in practice, while a refactoring can affect the execution speed, it usually does not affect the algorithmic complexity.
Semantic equivalence is only enforced within the point of view of a single thread. This is relevant as the Java language (and most other existing languages) explicitly provides several facilities when two or more threads need to manipulate the same variables at the same time, using explicit synchronization. It is in fact considered as a very bad programming style (if not simply as incorrect) to make multi-threaded code that depends on the exact execution order of the threads [71]. With this assumption, the presented refactoring algorithms are also applicable to properly written multi-threaded code.

It is also assumed that a refactoring is always applied on code that has no compile error. Semantic preservation obviously implies that the resulting code must also compile without errors.

Any refactoring has preconditions and postconditions [58, 74, 77]. A direct implication is that there are various cases in which a given refactoring is not possible (when the preconditions are not satisfied). Hence, a refactoring implementation must either modify the code as specified by the refactoring definition, or leave it unmodified and report an error to the user.

Finally note that it is not our goal to determine whether or not two arbitrary codes are semantically equivalent. This is indeed a problem that is not decidable [81]. We restrict ourselves to comparing semantics of a given code and a modified version of it. In the context of refactoring, the equivalence between the two versions is deduced from the fact every single transformation that was applied to the initial code is semantics preserving.

### 1.2.2 Definition of the Context

According to the above definition, refactoring is a process that transforms source code to semantically equivalent source code. Although this definition is sufficient to understand the algorithms that are addressed in this thesis, it is important to also understand the context in which the algorithms are meant to be used.

Indeed, depending on the context, it may be necessary to tune the same algorithm in different ways, such as speed versus accuracy, approximations versus exact, etc.

Transforming source code in general can be used in the following different contexts:

**Design improvement** This is the main goal of refactoring. The idea is to improve an existing design, either because it is suboptimal, or because it has to be adjusted before new features are added. This was reviewed in Section 1.1.

**Design recovery** Design recovery is concerned with legacy, and usually badly written code, whose authors are no longer available. Design recovery attempts to automatically transform unstructured code into structured code, so that the resulting code is easier to understand. A typical example of design recovery is the replacement of `GOTO` statements by structured loops in legacy COBOL code. Clone detection and extraction also fall in this category.

**Optimization** Optimization aims in improving the performance of the code, either in term of execution speed, algorithmic complexity or memory usage. Although many optimizations can be done by the compiler, some refactorings can also be used to improve performances. Examples are "inline method", "reduce scope of variable", "remove middle man", etc [28].

In this thesis we are focused on design improvement. This has the following implication on the proposed algorithms:

---

2 The process of finding code duplications and removing them. This is reviewed in details in Section 2.2.3.
User interaction is favored against automatic decisions. All decisions that are difficult to take (because the optimal choice is difficult to infer) are better left to the user. In particular, unlike design recovery, the code to transform and the transformation to apply are always considered as user inputs.

Clarity and predictability of the result is favored. This differs from optimization which can produce code that is difficult to understand.

Easy to understand results. The algorithm usually avoids doing more than what is requested, because it is typically used by the author of the code. Too complex transformations may confuse the user rather than help him. This differs from design recovery, which attempts to make code written by other persons (and hence mostly unknown by the user) easier to understand.

1.2.3 Definition of our Focus

It is time to differentiate two aspects of the source code of an application: the declarative part and the executable part.

The declarative part is basically the “structure” of the application. It is formed by packages and the class structure, the class hierarchies, the fields, the method signatures, etc.

The executable part on the other consists of all the method bodies.

Most of the research in the area of automated refactoring is mainly concerned with the declarative part [14, 58, 59, 74]: renaming and moving fields, methods and classes; pulling up and pushing down methods. These refactorings have usually little to perform on the executable part.

This thesis on the other hand focuses on refactorings that have to deal primarily with the executable part of the source code. Such refactorings are “extract method”, “reverse conditional”, “form template method”, “create method object”, etc. This is why we speak about refactoring statements. Indeed, in this thesis we are deeply concerned with the statements that form the bodies of methods, and only little concerned with the declarative part.

Finally, our target programming language is Java. While most of the material also applies to automated refactoring of similar languages, we do not address languages with major differences, such as dynamically typed, non object-oriented, or non imperative languages.

1.3 Automated Tools

Motivations

In this section, we discuss in more details the properties that are expected from an automated tool that implements refactoring transformations.

There is an unlimited number of ways of improving an existing design. In many cases, improving a design is a long, continuously refined process; and it is not a well-defined process, as it depends on the actual application. Nevertheless, it is believed that most refactorings can be decomposed into small steps, some of which are clearly defined and recurrent.

This has led various authors to propose catalogs of common refactoring techniques [27, 28, 47, 65]. Many of them have already been implemented in existing development environments [30, 59]. Despite the existence of such catalogs, it is commonly admitted that the modification of an existing design rarely requires the application of just a single refactoring, but usually involves several different refactorings. Based on this observation, it is relevant to consider small and simple refactorings, as long as they can be used one after the other to perform the desired changes [27, 49, 58, 77].

Among the small refactorings, some of them are frequently used, such as renaming items (variables, methods, classes, packages, etc) or extracting methods. While their definition is simple, they can be
cumbersome to perform manually. Renaming an item for instance, implies that every occurrence must be
looked for and renamed. Looking for and renaming all occurrences cannot simply be done by a textual
search-replace: there can be several items with the same name, yet with different scopes and hence
different meanings. Manually identifying whether an item with the same name is actually the same item
or not is hence error-prone.

As another example, extracting a method requires one to identify all the local variables that are read
and written within the extracted code, in order to pass them as arguments or result of the extracted method.
This is again an error-prone task to do manually.

Properties

The goal of semi-automated tools is precisely to perform such cumbersome work automatically, and to
leave only the main decisions under user control. The following properties are expected from such a tool:

- Determinism. A semi-automated refactoring tool rarely takes decisions by itself when multiple
alternatives are possible. For all operations that require a decision, some user input is asked for
and used. With our two refactoring examples (renaming and method extraction), this concerns for
instance the new name of the item to rename or of the method to extract, and of course the item
to rename itself or the code snippet to extract. In case some decisions are still taken by the tool,
they should be taken in a predictable manner. For instance, when extracting a method, the tool may
either ask the user for the names of the new method’s parameters, or generate predictable names
automatically such as arg0, arg1, etc.

- Correctness. The tool must either perform the refactoring correctly, meaning that the transformed
code has the same semantics as the original one (according to the definition of section 1.2); or leave
the code unmodified and report an error explaining why the refactoring cannot be performed.

1.4 Highlight of our Contributions

Several tools already exist for the automation of refactoring [14, 30, 58, 59], and the underlying technol-
ogy is reviewed in Chapter 2. However, this thesis aims in addressing various limitations of the underlying
methodologies (these are further developed in Chapter 3):

- Limitation to the declarative part. Most of the existing models and algorithms perform well in
refactoring the declarative part of an application. However, these models are poor when applied
to the executable part, such as the body of a method. While several models also exist for the
executable part, they are mainly targeted to the compilation process and are therefore suboptimal
or just inadequate for refactoring.

- Limitation to simple refactorings. The transformations that are actually implemented in existing
development environments are only a subset consisting of the simplest refactorings that are pro-
posed in the literature [27, 47]. As a consequence, the underlying theory is not ready to use for
more complex transformations, as shown in Chapter 3.

- Buggy implementations. Although some rare IDEs\(^3\) (such as IntelliJ) perform most refactorings
nearly correctly, the average IDE provides implementations that have various bugs [18, 42, 77]\(^4\).
Therefore, despite of the powerful models and algorithms, it seems there is still a phase shift be-
tween theory and practice.

\(^3\)IDE = Integrated Development Environment, such as Eclipse, NetBeans or Visual Studio
\(^4\)Several additional examples of refactoring bugs in existing IDEs are illustrated throughout this thesis.
In this thesis, we hence tackle the problem with a case study that helps us to explore all the above limitations. The chosen case study is the “form template method” refactoring, including all the subtransformations it involves: code comparison, data and control flow analyses, method extraction, and pulling up a method.

Through this case study and its resolution, we offer the following contributions, that directly address the above limitations:

- New models and algorithms that cope directly with the executable part of an application: the body of the methods, or simply said, the statements. Both new techniques and adaptations of existing techniques originating from the fields of compilation and clone extraction are proposed.

- An implementation of a refactoring that is more complex than what is usually found in existing IDEs. This refactoring not only seeks for new algorithms, but also exhibits new properties, such as the fact it can transform the code in several ways that are all correct according to its definition, or the fact it involves several other refactorings in its process.

- Algorithms and models that are more adapted to the task. While existing algorithms used for compilation or clone extraction could be reused, they show to be more complex than what is required, or simply inadequate for the refactoring process. Simpler and faster approaches are proposed, that potentially permit easier implementations.

More precisely, we introduce the following new techniques:

- A new algorithm for the detection of duplicated code, and an extension of the algorithm that performs a differentiation between two code fragments. The proposed algorithm is based on the Abstract Syntax Tree, it abstracts from irrelevant information such as syntactic sugar and white space, yields no false positives and has a near-linear complexity.

- A new algorithm to analyze the data and control flows with respect to the process of extracting a method. The proposed technique is specially targeted to refactoring, and is therefore simpler than those used in compiler optimization. It only requires the Abstract Syntax Tree and runs in linear time.

- A systematic way of automatically resolving the preconditions of method extraction that are not satisfied during the process of forming a template method. In particular, we discuss the problem of extracting a method that would need to return more than one value, a feature that is not directly available in Java. We give an extensive overview of existing solutions, and propose a new one that is specifically targeted to the proposed case study.

Our goal is to provide a fully working implementation of the “form template method” refactoring on Java code. Hence we also describe the existing tools on which we rely, discuss in details the new models and algorithms we propose (both in terms of the high level approaches and low level implementation details), and illustrate how the resulting algorithm performs on various examples of Java code. The implementation of the refactoring we are going to present is written in Java, and makes use of the plugin architecture of the Eclipse IDE [39].

1.5 Structure of the Document

For this thesis, the following methodology has been adopted:

- In a first step, an overview of existing techniques is proposed (Chapter 2).
• In a second step, limitations of the existing techniques are reviewed. To illustrate these limitations, we use our case study and show that it cannot be solved automatically with state of the art methodologies (Chapter 3).

• In the next steps, which are the main contributions of this thesis, we propose extensions of the existing techniques and new techniques to solve the chosen case study (Chapters 4 to 6). We also discuss how the proposed methods can be applied to other problems.

• Finally, Chapter 7 briefly explores other and different areas of research that share some similarities with our work.

More precisely, the remainder of this document is structured as follows: Chapter 2 reviews existing models and techniques for code analyses and transformations. Chapter 3 illustrates the “form template method” refactoring as a case study, and shows that it seeks for new methodologies.

Chapter 4 deals with the problem of comparing two source code snippets, and gathering their differences and similarities, in a way that is more clever than a text comparison, by taking care of the underlying language’s syntax. Chapter 5 deals with the problems of data and control flow analyses, from which important information is extracted from the source code in an automated way. Chapter 6 finally proposes a systematic way of trying to resolve preconditions that are not satisfied in the middle of the transformation process.

In Chapter 7, we give an overview of other research fields that are not directly related to refactoring, but share some common methodologies. We conclude in Chapter 8.
Chapter 2

State of the Art

In this chapter, we present the state of the art in the field of refactoring implementations. Section 2.1 presents different data structures that are used to represent source code in ways that make refactoring easier to implement. Section 2.2 presents existing algorithms that are used to analyze and transform source code.

2.1 Modeling Source Code

In this section, we discuss commonly used representations of the code. An obvious representation is just "the" source code, as plain text. While this representation has some advantages (some are pointed out in Section 2.2.3), it usually does not provide enough abstraction for the implementation of complex transformations. Consider as an example two different refactorings: renaming a variable, and extracting a method.

When the source code is represented as plain text, implementing these refactorings may seem to be easy at first glance: renaming a variable looks like a search-replace operation, and extracting a method looks like a cut and paste operation (with a few additional code generation, such as the new method’s signature). In fact, this may work for simple cases, and indeed some people have successfully applied refactoring on less simple cases by using some regular expressions in addition [28]. Nevertheless, in the general case, a text-based transformation cannot easily handle several problems such as the following:

- Scoping. A variable has a given scope in which it is visible and accessible. A field for instance is visible in the class it is defined, and optionally in subclasses and/or other classes of the same package, depending on its defined visibility (private, public, protected, etc). A local variable is only visible within the block in which it is defined (method body, loop body, etc). At any place where a given field or variable is not visible, it is possible to define another variable with the same name. In such a case the two fields or variables are distinct and each one has its own purpose. A simple search-replace is not able to distinguish them. But in practice, we usually only want to rename one of them.

- Name conflicts. When we rename a variable to a given target name, we first have to look for fields or variables with the same target name. The renaming can only be done if the two variables do never occur in the same scope. In the latter case, the renaming might be possible in some situations, but the occurrences of one of the two variables might need to be changed. For example, if we rename a method variable foo into the target name bar, and the class already contains a field named bar, the renaming is possible, but occurrences of the field within the scope of the variable must be prefixed
this.bar. On the other hand, if the method already has an argument named bar, the renaming is not possible.

- Data flow. When extracting a method, just cutting and pasting the desired portion of the code into a new method might not be sufficient. If the portion of code accesses variables defined before it, or changes the values of variables that are accessed after it, it is necessary to identify these variables and to pass them as arguments and results of the extracted method. This also implies that their data types must be identified.

- Control flow. Arbitrary code cannot always be extracted without first applying some transformations on it, and sometimes cannot be extracted at all. It is not possible for instance to directly extract a portion of code containing a break statement without the corresponding enclosing loop.

In Section 2.1.1 we discuss the Class Graph, and in Section 2.1.2 the Abstract Syntax Tree (AST). These two representations are complementary in that they each model a different aspect of the code, and they can together fully and effectively\(^1\) model arbitrary code. They are the most commonly used representations of the source code, not only for refactoring, but also as a part of the compilation process.

In Sections 2.1.3 to 2.1.6 we present models that only give a partial representation of the code and are also more difficult to build from the source code. Yet these models are the closest to the actual semantics of the code and are hence of great importance for many analyses and transformations. Other less used representations are discussed in Section 2.1.7.

Note that in the remainder of this document, we use the expression “source code” to refer to the textual (or string) representation of the code. We use the term “code” alone when speaking of the code regardless of its representation (textual, AST, Class Graph, etc).

### 2.1.1 The Class Graph

The Class Graph [58, 59] models classes, fields, methods and their relations. It hence captures the high-level structure of the code. An accurate way of defining it is that it models all aspects of the code which are not “executable”, that is, about everything except the bodies of methods.

The Class Graph is closely related to UML class diagrams with a major difference though: UML is general-purpose, whereas the Class Graph is specific to a given programming language. UML captures notions that may not have a direct equivalent in the programming language. An example is one-to-many relations which may require the usage of a list data structure in the target programming language. The Class Graph is closer to the target programming language, and the presence of a list for instance is just modeled as such, and is not interpreted.

#### Nodes and Edges

Formally, a Class Graph is a graph whose nodes and edges are augmented with additional information.

A node of a Class Graph represents one of the following:

- A class or interface
- A field
- A method
- A parameter of a method

---

\(^1\) Strictly speaking, the AST alone is sufficient to model arbitrary code, but can be very ineffective for some tasks and is hence commonly used together with the Class Graph.
Eventually, a package might be represented as well. A node is hence defined by what it represents and by the name of the represented item.

An edge of the graph can represent various kind of relations between the nodes, such as:

- Superclass and subclass relations
- Container relations between a class and its methods and fields, and between a class and nested classes
- Container relations between a method and its parameters
- Type relations between a field and its class
- Type relations between a parameter and its class
- Access relations: whether a given method reads and/or writes a given field
- Whether a method possibly invokes another method. We say “possibly” as we cannot always determine the exact methods that will be invoked at run-time, in the presence of conditionals statements or dynamic bindings
- Whether a method possibly throws an exception of a given class

Depending on the transformation or analysis we want to perform, not all kind of relations are necessary. Language-specific constructs may require additional node or edge types. Examples are generics (since Java 5) or anonymous classes defined within a method’s body. Unlike UML, there is usually no definite list of what is modeled by a Class Graph, as it heavily depends on the underlying programming language.

In many cases, it is necessary to put additional information in the Class Graph. This is typically done by adding attributes on the nodes and edges. Attributes can for instance be used to model modifiers (public, private, final, transient, abstract, etc).

It should also be noted that some kinds of relations do not make sense for some programming languages. Edges expressing the type of a given parameter for instance are useless when modeling a dynamically typed language.

Figure 2.1 shows an example of Class Graph. The example corresponds to the modified class diagram of Figure 1.1. However, not all items are displayed due to lack of space.

Note that relations such as “invokes”, “accesses” and “updates” can only be deduced by analyzing the body of the corresponding method. The Class Graph implementation provided by the Eclipse Java Development Kit does not model these relations. They are also not used by the different algorithms presented in this thesis. Instead, these relations, when required, will be fetched from another representation of the code, the Abstract Syntax Tree (see Section 2.1.2).

The Class Graph has a wide use in refactoring implementation, although it does not need to be modeled explicitly. The Eclipse JDT for instance uses regular Java objects in place of explicit nodes, and references between objects in place of explicit edges. This is sufficient for simple analyses and transformations [30]. For more complex refactorings, the graph is modeled explicitly and graph transformations (discussed in Section 2.2.1) are used [58].

**Typed Graph**

It is clear that not all Class Graphs are valid representations of source code. Programming languages have various restrictions on the kind of nodes and edges that are permitted. For that purpose, it is possible to define a type graph for a given programming language and to check whether a given Class Graph conforms
to the type graph [58]. The notion is similar to the notion of well-formedness in model engineering, and to the notion of conformance of an XML file with a given XML schema.

A type graph for instance can define whether the language allows multiple or only single inheritance. It also defines what items can be nested. In Java for instance, a class can contain methods and nested classes, but a method can only contain nested classes. Other languages such as Modula-2 allow methods to be nested.

Related Models

The Class Graph is close to an UML class diagram. However, the Class Graph is strongly coupled with the underlying programming language, whereas UML class diagrams aim to be language independent. Furthermore, UML diagrams in general are targeted to comprehension and documentation, but are hardly useful for refactoring.

However, there have been successful attempts to create models that are both mostly language independent and able to perform refactoring transformations. One example is the FAMIX model [22, 74], proposed as a replacement of UML, and which can accommodate several programming languages. The FAMIX model is closer to the Class Graph than UML class diagrams. It is however mainly targeted to round-trip engineering: converting the source code into the FAMIX model (reverse engineering), and converting back the model into source code (forward engineering). In this scenario the model is not just an internal representation used temporally to implement a refactoring, but an alternative view of the code with which the user can interact (structure understanding, calculation of metrics, code smell detection, refactoring, etc).
2.1.2 The Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a representation of the source code that is commonly used in compilers. An AST is a full representation of the source code in the sense that it is possible to re-generate an equivalent version of the original source code from an AST. The only things that are not modeled by an AST are spaces, blank lines and comments, although we will see in Section 2.2.4 that the AST can be augmented to deal with these in a non intrusive way.

The AST is closely related to the parse tree and hence to the formal grammar of the programming language. The only difference with the parse-tree is that the AST usually removes useless constructs, such as useless parentheses. The rules on what and how useless constructs are removed are not clearly defined and vary from one implementation to another.

Figure 2.2 shows the AST corresponding to the following code:

```java
import static java.lang.System.out;

public class HelloWorld {

    private static String text = "Hello world!";

    public static void main(String[] args) {
        int count = 5;
        while (count > 0) {
            out.println(text);
            count--;
        }
    }
}
```

There are a few things to note about the AST:
• There is a finite number of node types. The different node types correspond to the different constructs of the underlying programming language (declaration, method invocation, all basic operators, etc).

• Node types are usually implemented by classes with a suitable hierarchy. For instance, the node type for a constant and the one for a variable access usually both extend some abstract node type modeling expressions.

• Each node type has predefined properties. The “Method declaration” node for instance has properties for the method name, the modifiers and the return type.

• Each node type has a predefined set of possible child nodes and child node types. The “Method declaration” node for instance accepts exactly two child nodes: one for the parameter list, and one for the body. The “Block” node (modeling any sequence of consecutive statements within { braces }) accepts any number of child nodes corresponding to the statements.

Depending on the actual implementation, some elements of the AST might be modeled as node attributes in one implementation and as child nodes in another one. In the above example, we modeled identifier names as attributes. In the implementation provided by Eclipse, all identifier names are modeled as child nodes; this allows the addition of attributes on the name itself, such as a reference to the corresponding node in the Class Graph.

While an AST potentially contains all the information that can be given by the Class Graph, that information is not expressed in a straightforward way and is hence more difficult to find out. Consider as an example the relation between a method and the invoked methods. The AST for instance contains all the method calls performed by the body of a given method, but it is necessary to traverse the tree in order to get them. The Class Graph on the other hand provides direct access to such information.

Regarding scoping, an AST usually only contains the names of variables, fields, classes and methods. Hence it is again necessary to analyze the whole AST in order to differentiate between two different items (i.e. two variables with different scopes) that have the same name. Finding the scope of variables, fields, methods and classes occurring in an AST consists in creating the Class Graph, and then linking nodes of the AST to the corresponding nodes of the Class Graph when relevant. The link between an AST node representing a field, variable, method or class to the corresponding node of the Class Graph is sometimes named the binding information [30].

Because the binding information is important for both compilation and transformations, it is common to use both the AST and the Class Graph at the same time, and to link nodes of the AST with nodes of the Class Graph.

A straightforward use of the AST is pretty printing. Pretty printing consists in correcting the indentation and normalizing the use of white space and new lines according to some predefined coding style. Pretty printing is easily implemented by parsing the source code into an AST, and then regenerating the source code from the AST. The white space and new lines are inserted during the regeneration of the source code using simple rules, such as the block nesting level, or the length of the current line. In modern development environments, the exact rules can be parameterized, in order to get a result that conforms to a given programming style.

Throughout this thesis, we use a simplified AST representation, in which attributes are not shown, and identifier names and constant values are placed directly on the corresponding node\(^2\). Figure 2.3 is a simplified representation of the AST of Figure 2.2.

---

\(^2\)There is no formal definition of how an AST must be illustrated, and several different informal representations are used in the literature. What is actually important in practice is the underlying data structure.
Figure 2.3: Simplified representation of the Abstract Syntax Tree of the “Hello World” program.

2.1.3 The Control Flow Graph

The Abstract Syntax Tree described in the preceding section is a model that is very close to the source code. Indeed, it exhibits a one-to-one mapping with the source code, and hence provides a complete representation of it. The Class Graph on the other hand, only models a particular aspect of the code, and some of the modeled aspects are not available in a straightforward way from the source code or from the AST.

The Control Flow Graph (CFG) also focuses on a particular aspect of the code which is not trivial to extract from the source: the possible execution paths at runtime. A CFG hence models dynamic aspects of “executable” code, such as the body of a method. Figure 2.4 illustrates the CFG of the following source code:

```java
public void gcd() {
    int a = getA();
    int b = getB();
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    setResult(a);
}
```

In a CFG, a node is one of the following:

- an assignment
- a void method call
• a conditional (if, switch)
• a loop predicate
• a “jump” instruction (break, return, continue)

Non-void methods, or function calls, are usually not represented in their own node, but are rather merged with the node of the operation in which they are used (assignment, conditional, etc).

The CFG makes it easy to analyze execution branches (due to conditionals) as well as loops, modeled as cycles in the graph. For convenience, a CFG is typically augmented with two special nodes, an entry and an exit node. The entry node (“START”) is used to model the “begin” of the graph, that is, the point at which the execution flow starts. The exit node (“END”) is used as an endpoint of the edges that leave the modeled code.

The CFG is essential in compiler optimization and in analysis tools. A common use of the CFG is to check for reachability. If a part of the graph is disconnected from the entry node, the corresponding code is unreachable and can be safely removed without affecting the semantics. Such code is called dead code. Note that in general not all unreachable code can be detected (halting problem).

If the exit node in not reachable from the entry node, the code has an infinite loop. Again, not all infinite loops can be detected. However, unreachable code or infinite loop that are detected as such are detected with 100% accuracy (without false positives).

Note that a CFG is close to a flowchart such as an UML activity diagram. However, the CFG is restricted to method bodies or body fragments, whereas flowcharts can also model interactions between methods and classes. Furthermore, the CFG is built from existing code whereas flowcharts are usually documentation artifacts.
The CFG is not directly used for refactoring transformations, but it can be used to check for some preconditions of complex refactorings such as method extraction [77]. An example is the fact that a method cannot be extracted if it contains more than one return path (such as return statements, or break statements without the escaped loop). This is reviewed in Section 5.5.

2.1.4 The Data Flow Graph

Whereas the CFG models the flow of execution, the Data Flow Graph (DFG) models the flow of data.

Figure 2.5 shows the DFG corresponding to the following code:

```java
public void gcd() {
    int a = getA();
    int b = getB();
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
    }
    setResult(a);
}
```

In a DFG, nodes correspond to instructions and operators, whereas edges correspond to data, or variables. Edges are only present if the underlying variable’s value has to be “transmitted” from one node to another, meaning that the target node uses the value created or modified by the source node.
Observe that the DFG does not model the execution order, unless there are data dependencies between different operators or instructions. Like the CFG, the DFG is not a full representation of the code. In particular, the DFG does not clearly show the outcomes of conditional statements.

The DFG is useful for the extract method refactoring. One step of this refactoring is to identify the required arguments and results of the method to extract. Given the sub graph of the DFG corresponding to the statements to extract, this problem simply consists in looking for the incoming and outgoing edges of this sub graph.

In the given example for instance, the DFG immediately reveals that extracting either of the “%” expressions requires both a and b as argument, but returns only one value. Extracting the “setResult” statement requires only a as argument and returns no result.

The concept of DFG has a widespread use in the field of data flow languages. The idea of data flow languages was to abstract from the “sequential” paradigm, and to use a data-driven paradigm instead. There is no definite definition of a data flow language, but they fall into the following main categories:

- Visual languages. In these languages, the DFG is the main representation of the code. ProGraph is an example of such a language [61]. In ProGraph, everything is visual. Hence a DFG is augmented with other information such as the class structure. Fabrik is another example [40]. In Fabrik, the DFG is only used for high-level interaction of GUI components, and a regular language such as C is used for the rest of the code. An interesting aspect of Fabrik is that a GUI component is an integral part of the DFG (actually each GUI component is a node).

Visual languages are widely used in some specific domains where data is the most important aspect. An example is the field of signal processing and in particular audio processing. Max/PSP and Soundium [63] are two examples of data flow languages targeted to audio processing.

Not all visual languages are based on the concept of the DFG, but some of them share some similarities. Examples are Petri nets, and the “G” language of LabView.

- Parallel and Distributed languages [32].

- Spreadsheets are considered as a kind of data flow language, as the actual computations are triggered by the introduction or modification of the data.

- Event-based languages can be viewed as sorts of data flow languages, if we consider that an event is just some data. Various programming languages are including event-based notions (Visual Basic, Java, C#), but they usually rely on the standard imperative and sequential paradigm.

### 2.1.5 The Program Dependence Graph

The Program Dependence Graph (PDG) models both data flow and control flow dependences. Unlike the CFG, but like the DFG, the PDG abstracts the execution order.

Figure 2.6 shows the PDG of the following code:

```java
public void gcd() {
    int a = getA();
    int b = getB();
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    setResult(a);
}
```
int a = getA();
int b = getB();
while (a != b)
setResult(a);
if (a < b)
    a = a - b
    b = b - a

Figure 2.6: Example of a Program Dependence Graph (PDG).

In a PDG, nodes are instructions, like in a CFG. A control dependence edge means that the source node has an influence on whether or not, or on how many times the target node will be executed. A data dependence edge means that the value of some data created or modified by the source node will potentially be used by the target node.

Observe that a data dependence edge can only start from a node corresponding to an instruction that performs a write access (such as an assignment). Conversely, a data dependence edge can only end on a node corresponding to an instruction that performs a read access of the corresponding variable. A control dependence edge can only start from the entry node or from a node corresponding to an instruction that affects the control flow (or execution order): conditionals, loops and jump instructions (such as break and return statements).

An advantage of the PDG is that its edges only depend on the actual flows of data and on the nesting of blocks, but not on the order in which the instructions occur in the source code. If two instructions have no data or control dependences, it is possible to swap them without affecting the semantics. This is frequently used in compilers as a part of other optimizations. The PDG makes it easy to find out such independent instructions.

The PDG can also be used for Slicing, a powerful technique that is described in Section 2.2.2.

2.1.6 FOOD: a Data and Control Flow Model

With the exception of the Abstract Syntax Tree (AST), the models discussed in the previous sections do not provide a full representation of the code, but only a partial view focused on a particular aspect. However, while the AST provides a full representation, it does not provide the same level of abstraction regarding data and control flows compared to the other models.

The FOOD (First-class Object Oriented Dataflow) model was developed early during this thesis as an attempt to provide a model that has both advantages: good abstraction of control and data flows, and full representation of the code. FOOD consists of both a Class Graph and one or more execution graphs similar to the PDG, in that they exhibit both data and control flows. A property of a FOOD execution graph is that it is executable. An interpreter has been developed, that allows FOOD execution graphs to
be executed [44]; the interpreter allows full access to the Java API classes and methods from an execution graph.

Figure 2.7 shows the FOOD graph of the sample code replicated three times in the previous sections. Observe that the graph is close to a DFG, with the following extensions:

- Control flows are additionally modeled;
- The fby (first, followed by) construct is explicitly modeled as an instruction of the FOOD language;
- Instructions (represented by boxes) have a fixed number of inputs and outputs, illustrated by input and output ports.
- Each input port is connected by exactly one data flow edge coming from an output port. Output ports can be disconnected.
- Output ports can be connected by control flow edges; a control flow edge connects an output port directly to an instruction.

The execution of a FOOD graph is flow driven, and is implemented by a simple iteration: it repeatedly executes (possibly in parallel) all instructions that have data on all their input ports and that have been triggered by all their incoming control flow edges. After an instruction is executed, the inputs are cleared and the results are sent to the connected instructions. Two special cases to note are that:

- FBY is a special instruction; it can be executed when only one input has been received. Actually its first execution requires only the first input and all subsequent executions require the second input only. Methods in FOOD can have similar behaviors if their implementation is a graph containing an FBY instruction.
• If only triggers one of its outgoing control flow edges, depending on the boolean input argument.

Let us detail the process of executing the FOOD execution graph of Figure 2.7. In the first iteration, `getA` and `getB` are executed because they have no input and no incoming control flow. Their results are propagated to the two `FBY` nodes that get executed in the second iteration. After that, “!” is executed as it has received its two inputs. “>”, `setResult` and the two “%” on the other hand are still waiting because they have not received any trigger from their incoming control flow edge. Then the first if is executed. It either triggers the “>” or the `setResult` node, and the corresponding node gets executed as it already has all other inputs. The program terminates when there is no more instruction to execute. This occurs when the `setResult` instruction is executed: as it has no outgoing edge, it does not trigger the execution of further instructions.

The fact that FOOD fully models the code also implies that transformations such as refactorings can be applied directly on the FOOD representation of the code. Both the “extract method” and “form template method” have been successfully implemented on this model [44]. The CFG, PDG and DFG on the other hand are usually only used for analyses, and the transformations are made on the AST.

Furthermore, by abstracting both data and control flows, a FOOD execution graph allows parallel execution whenever two or more different instructions have no data or control flow dependences.

While FOOD can be used directly as a programming language (an editor exists to directly build FOOD Class Graphs and execution graphs), another potential use is to implement transformations of another programming language such as Java. In such a use, the Java code to transform must be first transformed to the FOOD model, then the code is modified in the FOOD graphs, and finally the result is converted back to Java. As FOOD models data and control flows directly, complex transformations such as method extraction or forming a template method can be done quite easily within the FOOD representation of the code [44].

The construction of a FOOD execution graph from existing source code has some pitfalls though, that can be considered either as advantages or drawbacks, depending on the point of view and desired goals:

• The exact ordering of consecutive instructions with no data or control flow dependences is lost. This is an advantage for optimization and for various transformations, because it permits these instructions to be reordered in a way that is optimal for the given task. Instruction reordering for instance is very useful in compilers, for example to optimize the CPU pipeline’s usage. Instruction reordering can also be used in complex refactoring such as clone extraction: in some cases, some code can only be extracted in a new method when some of the instructions are reordered [51]. In case one wants to keep the exact ordering of the original source code though, the FOOD execution graph must be explicitly augmented for that purpose.

• A given source code can result in multiple different FOOD execution graphs, depending on how well the data and control flow are analyzed. Indeed, it is hardly possible to only keep the minimal number of data and control flow dependences; in practice, heuristics are used, and some superfluous dependences are added [24]. Note that this problem is shared by all other flow models such as CFG, PDG and DFG. On the other hand, it is always possible to make an “unoptimized” FOOD graph that is nevertheless semantically equivalent to the original code. In other words, getting a minimal graph is only a matter of optimization.

• Not all FOOD graphs correspond to valid Java source code. This can be a problem when transformations are performed directly on the FOOD model, yet this is resolved by the use of preconditions, which are a mandatory step in any refactoring implementation anyway.

• The construction of a FOOD model from existing source code is a tedious task that may not be worth the advantages. This is further discussed in Section 2.2.2.
2.1.7 Other Models

There are many other models of the source code used for other purposes similar to refactoring and it would be impossible to cover all of them. The gcc compiler for instance, makes use of a common intermediate language, before it generates code for a particular processor [29]. This intermediate language, named RTL (Register Transfer Language), allows most compiler optimizations to be performed in a way that is independent from both the original language and the target processor.

Apart from analyses and transformations, other models exist for different purposes. UML for instance provides several models for design and code understanding [26]. The Java byte code can also be viewed as a platform independent model of Java source code. Models themselves can be modeled, and this theory of “meta-models” is an active research area [8].

Finally, considering an application as an abstract process, the source code itself can be viewed as a model of that process [66], like the AST, diagrams, or any other representation. In particular, in this thesis we consider the source code as the only available model of the existing design that has to be refactored: all the representations discussed previously (AST, Class Graph, etc) are built from the source code as an integral part of the process. Refactoring implementations that also have to synchronize other readily available representations (such as UML diagrams) will not be addressed.

2.2 Analysis and Transformation Methodologies

The previous section has given an overview of common abstract representations of the source code. This section discusses existing techniques for the analysis and transformation of source code using such representations. In general, only the textual representation of the code, the source code, is readily available. Hence the previously discussed models have to be built from the source code before a transformation can be applied. However, some kind of analyses can be done with relative success directly on the source code, as will be seen in Section 2.2.3 for instance.

2.2.1 Graph Transformations

A graph is a well-known data structure, and allows manipulations through many well-known algorithms [17]. In the domain of refactoring though, the term “graph transformation” is almost exclusively used to denote a formal framework for transformations of the Class Graph described in Section 2.1.1. As the Class Graph represents the high-level structure of the source code (the declarative part), refactoring based on graph transformations are referred to as “model refactoring” in the literature [58].

While the CFG and PDG are graph structures as well, they are mainly used for analyses only, and transformation algorithms typically rely on another model such as the AST for the actual transformations. They will be discussed in Section 2.2.2.

Graph Productions

Transformations of the Class Graph (referred to as “graph transformations” in the remainder of this section), are defined, in the simplest case, by a graph production rule \( p : L \rightarrow R \) where \( L \) and \( R \) are graphs. \( L \) is the “pattern” whose occurrences in the Class Graph we want to transform. \( R \) is the transformed version of the pattern \( L \). If \( L \) is a subgraph of the Class Graph, applying the production rule \( p \) will transform the Class Graph by replacing all occurrences of the \( L \) subgraph by \( R \). If the Class Graph contains no occurrence of \( L \), the graph transformation just does nothing. The replacements of the \( L \) occurrences by \( R \) are performed only once, there is no recursive application in the case a new occurrence of \( L \) appears as a result of the replacements.
In practice though, graph transformations are done in such a way there is only one occurrence of $L$ in the Class Graph, which is also meaningful for representing a single refactoring at a time. In that case, $L$ can be viewed as a precondition of the refactoring. In various cases, a given refactoring can be applied under multiple different situations. For that purpose it is frequent to use more than one graph production rule to express a single refactoring. In that case, all production rules are executed, although in practice only one of them actually yields a transformation.

$L$ is named the left hand-side of the graph production rule, and $R$ the right hand-side.

There is a strong relation between graph production rules applied to graphs, and search-replace using regular expressions on text. Like in the later, graph transformations can use wildcards for the names of the nodes and edges in the left hand-side $L$.

**Negative Application Condition**

While graph production rules can express a variety of graph transformations, in practice it is sometimes necessary to use several production rules with large subgraphs $L$ and $R$ to express a simple transformation, especially when the transformation must only be applied in the absence of some specific pattern. An extension to address this problem is the introduction of a negative application condition (NAC) in addition to $L$ and $R$.

Whereas $L$ corresponds to a pattern that must occur in the graph for the transformation to be performed, a NAC corresponds to a pattern that must not occur. Multiple NAC can be attached to a single graph production rule. The graph transformation is only applied on occurrences of $L$ in which none of the NAC occur.

Figure 2.8 illustrates a graph transformation corresponding to the “pull up method” refactoring of method $X$ from class $B$ to an ancestor class $A$. The left hand-side $L$ is on the top left, and the right hand-side $R$ on the top right. On the bottom is an example of a NAC: it states that the refactoring is
not possible if there is an intermediate class $C$ between $A$ and $B$ in the class hierarchy, and that class $C$
already contains a method named $X$.

Further notions

Other topics related to graph transformations include, for instance, the composition of multiple refactorings. The notions of critical pairs and confluence for instance, provide a way of checking for potential conflicts between two transformations. Composition of refactorings also raises the question of getting the composed preconditions and postconditions. Further details on these topics can be found in papers by Tom Mens [58] and Günter Kniesel [49].

While excellent formalisms have been developed in the area of refactoring preconditions and post-conditions, the practice does not always follow the theory. Rather than trying to fully express all the precondition of a refactoring, the developers of the Eclipse development environment for instance have chosen a less elegant, yet simpler approach: most refactoring algorithms just “try” to perform the transformation on a temporary copy of the code, and then check if the result still compiles. If this is not the case, the refactoring is not performed on the actual code and an error is reported to the user. This is a way of using a post-condition (the resulting code must compile) as a precondition [39].

Limitations

The main advantage of graph transformations in general is that they provide a formal and mathematical framework that allows for various automatic proving of properties, including the possibility of proving that a given transformation is correct. On the other hand, their effective usability for non-trivial transformations is debated. In particular, while transformations of the Class Graph are easy to implement with graph transformations, transformations of the AST have been reported to very quickly introduce overwhelming complexity [58], despite the fact that an AST is a tree, and hence just a specific kind of graph.

Effectiveness

While graph transformations provide an excellent framework for formal proofs, their implementation is quite tedious. Although transformation tools based on graph transformations exist, they do not yet cross the Rubicon [28].

For most cases, it is believed that the formalisms provided by graph transformations (including the possibility to prove the correctness of a transformation) is hidden by the complexity of an implementation: a formal proof of correctness for instance is voided in the presence of bugs in the implementation. Because a complete implementation of a transformation using graph transformation would require more code than a simpler version based on the AST, it will also be potentially subject to more bugs.

For that reason, it is not uncommon to discover that refactoring implementations in existing development environments (Eclipse is one example, which was verified by the author as the source code is open) are based on simpler algorithms (typically using the AST), although such algorithms can not easily be proven to be correct. Yet this preference can also be explained by the fact the AST is readily available, as its construction is already part of most compilers.

---

3The expression “crossing the Rubicon” is used by Martin Fowler in his refactoring web site [28] to qualify refactoring implementations that are both (mostly) correct and beyond what are considered by him as trivial transformations. The “extract method” refactoring is given as “the” simplest non-trivial transformation; yet it does not seem to be possible to implement it using graph transformations alone [58].
2.2.2 Flow Analysis

Relevance in Refactoring

Flow analysis is the processing of analyzing the data and control flows of some source code. Such an analysis is particularly relevant in refactoring, especially for the “extract method” refactoring:

- The data flow determines the required arguments and results of the method to extract. In the case of the Java language, the data flow is also used to verify a precondition, namely the fact that the extracted method returns at most one result.

- The control flow determines whether the method can be extracted at all. Namely it must verify the single entry, single exit precondition: the execution can only enter the method at at a single place and leave it to a single place. This may not the case in the presence of break and return statements for instance.

An obvious way of performing the above tasks is to use one of the graph representations discussed in the previous sections: the CFG for instance directly shows the control flow, and hence checking the single entry, single exit condition is a trivial task. Similarly, the DFG immediately reveals the arguments and results of the method to extract. The PDG and FOOD execution graphs can be exploited for both tasks with the same ease of use.

Note that it is necessary to link components of these graphs (nodes and edges) to the actual source code locations, or to the AST nodes, or else the transformation cannot be done. This is discussed later in Section 2.2.4.

The use of data and control flow analyses for the “extract method” refactoring is reviewed in details in Chapter 5.

Building the CFG or PDG

While the data and control flow analyses and their use in refactoring look like simple problems using the CFG, PDG or other graphs, this convenience comes with a price that is unfortunately seldom mentioned in the literature: the construction of these graphs from the source code is a tedious task, both in term of programming complexity and algorithmic complexity. This typically involves algorithms that are iterative, and at least of \(O(n^2)\) complexity, if not of exponential complexity in some cases [25, 51, 76].

The exceptions are the AST and Class Graphs, whose constructions are fast and well mastered and are a part of any compiler. But as previously mentioned, they only provide a weak level of abstraction compared to the other models.

In other words, graph models such as the CFG or PDG allow easy implementations of complex analyses and transformations, but only if the graph is readily available. If both the construction of the graph and the analysis or transformation are taken together, the complexity of the whole task makes these models much less attractive.

While it remains a valid alternative for the flow analyses required by the “extract refactoring”, this graph-based approach has not been retained in this thesis, due to the complexity of building the corresponding graph. Instead, a new “lightweight” approach is proposed later in Chapter 5.

Program Slicing

Slicing is a direct application of the PDG. Slicing is not directly useful for the “extract method” refactoring according to the definition that is used in this thesis, which is to extract a fragment of consecutive statements. However, slicing performs a method extraction according to an other definition that proves to be useful in various practical cases: extract all the statements (consecutive or not) that affect the value of a given variable at a given location in the code.
Let us illustrate slicing on the following code:

```java
int sum = 0;
int prod = 1;
for (int i = 0; i < a.length; i++) {
    sum = sum + a[i];
    prod = prod * a[i];
}
System.out.println("Sum: "+ sum);
System.out.println("Product: "+ prod);
```

Unlike method extraction (according to our definition), the input for slicing is a single variable read access. The result of slicing according to the access to the sum variable in the before-last line results in:

```java
public int extracted1(int[] a) {
    int sum = 0;
    for (int i = 0; i < a.length; i++) {
        sum = sum + a[i];
    }
    return sum;
}
```

The result of slicing according to the access to prod variable in the last line on the other hand results in:

```java
public int extracted2(int[] a) {
    int prod = 1;
    for (int i = 0; i < a.length; i++) {
        prod = prod * a[i];
    }
    return prod;
}
```

By extracting two methods according to the two above slices, one can turn the original method into:

```java
int sum = extracted1(a);
int prod = extracted2(a);
System.out.println("Sum: "+ sum);
System.out.println("Product: "+ prod);
```

If the PDG of the code is known, slicing basically consists in reversing the edges and collecting all nodes that are reachable from the node corresponding to the access to the target variable [51].

Observe that:

- Slicing extracts statements that are not necessarily consecutive;
- When slicing using two different variables, some statements can occur in both slices. The for loop for instance gets duplicated in the above example.

In this thesis, we are mainly concerned with the extraction of methods consisting of consecutive statements. However, slicing is a transformation that can help in the presence of methods that would need to return multiple results. This is discussed later in Section 6.3.

Slicing is a popular topic in the literature of program analysis. Definitions, formalisms, properties and implementations are discussed in details in previous papers [24, 51, 83].
2.2.3 Clone Detection

The term “clone” is used in software engineering to denote code duplications. Code duplications are usually considered to be a bad practice, as they make code maintenance more difficult. Indeed, if a change has to be done, and concerns code that has been duplicated, all occurrences of the duplicated code have to be searched for and modified, which can be a tedious work especially when working on the code of somebody else. Failure to update one of the occurrences may not only lead to bugs, but to problems with the project’s management that are quite embarrassing: although a modification is supposed to have been done, the application sometimes still behaves like the old, unmodified version, because some clones were forgotten and not updated.

Clone detection is about the process of automatically finding all code duplications in an existing program. Usually, the idea is to not only restrict the search to code fragments that are syntactically equal, but to also find fragments that are syntactically different, but semantically equivalent. For example, it is desirable to find duplicated code even if some variables have been renamed in one occurrence.

In most cases, clone detection is followed by clone extraction, which consists in extracting the duplicated code in a new method (this is basically an application of the “extract method” refactoring), and replacing all the occurrences of the duplicated code by an invocation to that method.

Clearly, determining that two code snippets are semantically equivalent is not a decidable problem. Nevertheless, various clone detection tools exist. Although testing for semantic equivalence with 100% accuracy is not possible, clone detection tools use heuristics that still have enough accuracy for practical applications. In other words, clone detection shares similarities with the theory of search engines. In particular, the effectiveness of a clone detection tool can be measured by its recall and its precision.

The recall is the percentage of clones that are actually found among all the existing clones. Obtaining 100% recall is not possible if semantic equivalence is considered (and not only syntactic equivalence). The recall is less than 100% in the presence of true negatives: true clones that are not detected as such by the tool.

The precision is the percentage of true clones among those that are reported. The precision is less than 100% in the presence of false positives: code fragments that are reported as clones but that are not actual clones.

Clone detection is not limited to the detection of exact duplications (either in terms of syntax or semantics), but can also consider partial clones. These are clones with “gaps”, that is, clones in which the duplicated code still contains a few instructions or expressions that differ. Indeed, in such clones it usually remains possible to reduce the duplication by extracting the clone, and by “parameterizing” the differences. In simple cases, differences can be passed as arguments to the extracted methods. In more complex situations the strategy pattern or similar constructs might be necessary. Although targeted to a different goal, the refactoring discussed in Chapter 3, forming a template method, can be viewed as a generalized way of extracting partial clones. Hence the detection of partial clones could also be used to find potential candidates for that particular refactoring.

In general, clone detection itself is not directly considered as refactoring, but rather as “design recovery”. Design recovery differs from refactoring in that both the decision of what transformations to apply, and the transformations themselves are, at least partially, automated.

Nevertheless, clone detection methodologies can be used as integral parts of some refactorings:

- When extracting a method, a useful extension is to find clones of the extracted code, and to propose to the user to replace all of them by a call to the extracted method as well.
- When forming a template method, it is necessary to compare two snippets of code for differences and similarities, in order to keep only the similarities in the template method.
Clone detection can be used to automatically find potential candidates for the “extract method” refactoring. Likewise, detection of partial clones can be used to automatically find potential candidates for the “form template method” refactoring.

Several approaches to the problem of clone detection exist [78]. The main categories are summarized in the following subsections.

**String Based Approaches**

String based approaches search for clones using only the textual representation of the code. They are easily implemented using hash tables. The advantage is that they allow very fast implementations. The drawback is that the textual representation only allows comparison based on the syntax, but not on the semantics.

In general, even the simplest string based approaches at least attempt to ignore white space. However, string based techniques cannot (without modifications) find several kinds of semantically equivalent clones, such as:

- Clones in which some equivalent variables have different names;
- Clones in which independent statements have been reordered;
- Clones in which a construct has been replaced by an equivalent one (for example a while loop replaced by a for loop);
- Clones in which unreachable (or useless) code has been removed in one instance;
- Clones in which arguments of a commutative function have been swapped.

Because string based methods are easily fooled by small syntactical changes, they usually have a low recall. On the other hand they typically have a very high precision (close to 100%). Indeed, two code snippets that have the same string representation are very likely to have the same semantics. Note that there are still some exceptions, such as the following Java statement:

```java
x = a + b;
```

This statement can have different semantics, depending of whether the variables are strings or numbers. In the former case, two strings are concatenated, whereas in the latter case, two numbers are added. It is not possible to extract a function that handles both cases, at least not in a trivial way.

There are several ways to improve string based approaches. We first discuss parameterized string based approaches, and then token based approaches, which are both similar in terms of the global process, but differ in the implementation and representation of the source code.

**Parameterized String Based Approaches**

Parameterized string based approach go one step beyond “simple” string based approaches, by performing different preprocessing operations on the source code before the detection [23, 67]. Their goal is to remove or to abstract the code so that only the semantically relevant things remain. There are numerous operations that can be performed at this stage, such as:

- Removal of all irrelevant parts such as white space, new lines and comments.
- Removal of items of low relevance. “Low relevance” means that the item most of the time does not carry semantics. As such, the removal of such items can introduce false positives, but still improves the global effectiveness of the clone detection. More precisely, it increases the recall at the expense of the precision. Possible items with low relevance are, among others [67]:

- The \texttt{this} pseudo variable. In many cases, the absence of the \texttt{this} prefix in front of an expression is equivalent to the same expression with the \texttt{this} prefix.

- Block delimiters (typically braces). This is especially relevant as blocks with a single instruction can be written with or without braces in Java.

- Package prefixes. In Java, a class can either be referred to by its simple name (such as \texttt{HashMap}) if the package is in the import list, or by its \texttt{fully qualified} name (such as \texttt{java.util.HashMap}). In most cases, there is no semantic difference. An exception is for instance when \texttt{java.util.List} and \texttt{java.awt.List} have to be differentiated.

- Normalization. This consists in replacing an item by a more generic one. This also implies that different items might be replaced by the same one. Examples are [23]:
  - Normalization of literal constants of the same data type. Even if two different literal constants are present in two similar code snippets, these constants can be passed as an argument if the clone is ever extracted. Hence their difference is not significant. Literal constants are typically replaced by generic names (such as \texttt{p1}, \texttt{p2}, ...) according to their order of occurrence.
  - Normalization of variable names. It is frequent for variable with a different name to have the same meaning. Hence variables can be replaced by generic names (such as \texttt{v1}, \texttt{v2}, ...), like for literal constants.
  - Normalization of primitive types. In practice, the primitive types \texttt{short}, \texttt{int} and \texttt{long} can in most cases be replaced by \texttt{long} without affecting the semantics. Similarly, \texttt{float} can be replaced by \texttt{double}.
  - Normalization of syntactic sugar. For instance, \texttt{x++}, \texttt{x+= 1} and \texttt{x = x + 1} can all be replaced by a common expression. Similarly, \texttt{while(true)} and \texttt{for (;;)} are also equivalent and can be replaced by a common expression.

Some of these operations clearly improve the accuracy of clone detection, such as the removal of white space. But most operations yield an increase of the recall at the detriment of a reduction of the precision. A little reduction of the precision is acceptable, especially if we are interested in the detection of partial clones as well. The choice of whether or not a given preprocessing transformation has to be included in a clone detection tool depends on its average effect on the global performance of the algorithm (recall and precision). Refer to M. Rieger [67] for more.

**Token Based Approaches**

Token based approaches are very similar to parameterized string based approaches, but differ in the implementation. They also use preprocessing steps in order to remove irrelevant stuff and to generalize items that are most likely to be equivalent.

Unlike string based approaches, token based approaches first transform the source code into a list of tokens. Hence they require a \texttt{lexer}. Preprocessing operations are similar to those of parameterized string approaches, but are applied on the resulting token list. Similarly, clones are then found by searching similar sublists in the token list rather than in a string.

By using a lexer, token based approaches are more resistant to some artifacts that may occur with parameterized string based approaches. For instance, removal of the \texttt{this} keyword (a preprocessing suggested above) using the string representation of the code may also remove the word “this” in a string literal. This does not occur with token based approaches. However, token based approaches need to know the grammar of the underlying language precisely in order to create the token list. Hence they are harder to adapt to different programming languages than string based approaches.
AST Based Approaches

Clones can also be detected using the AST. With such an approach, detecting code duplication consists in detecting two similar subtrees in the full AST of the source code.

Like token based approaches and parameterized string based approaches, AST based approaches are insensitive to irrelevant information such as white space, but also to constructs that are less trivial to detect, such as unnecessary parentheses or braces.

Furthermore, AST based approaches provide a simple way of detecting partial clones, namely by comparing subtrees only up to a certain depth. Indeed, the deepest nodes in an AST are usually nested expressions that can easily be passed as parameters in the process of clone extraction.

In case the binding of the variables are resolved (when the Class Graph is built and linked to the AST, see Section 2.1.1), the AST allows further improvements: namely it can properly match identical members accessed in different ways (for instance with or without prefix such as fully qualified names or this), and can also properly differentiate different variables that have the same name.

The AST on the other hand, does not directly permit the improvements achieved by the use of normalization. However, the AST can be normalized as a preprocessing step (like with token based approaches) before the clone detection if desired.

Clone detection using the AST requires searching for similar subtrees. This is more complex than looking for similar substrings in a text representation, or for similar sublists in a token list. Hence, getting high performance is more challenging. Authors [5] have shown the problem to be of $O(n^3)$ complexity in general. They proposed a heuristic that builds a hash code for every subtrees, and then compare the hash codes in order to find potential clones. Their algorithm runs in $O(n^2)$ worst-case complexity, and still achieves good accuracy. In particular, they proposed to include a normalization process within the computation of the hash code: for example, the hash code corresponding to an expression using a commutative operator (such as an addition) can be computed in such a way that it does not depend on the order of the operands. Hence two clones in which operands of a commutative operator are swapped can still be detected.

Graph Based Approaches

Using the DFG, CFG or PDG for clone detection can result in very effective algorithms, but also introduces a lot of complexity [51]. These representations are complex to build from the source code, and as they are only partial representations, they need to be augmented in order to build an effective clone detector.

On the other hand, only graph-based approaches are able to detect “difficult clones” in elegant ways, such as [24, 52]:

- Clones whose statements are not consecutive;
- Clones with independent statements whose order is different;
- Clones consisting of intertwined statements.

This is a direct consequence from the fact that graph representations such as the DFG or PDG abstract from the actual statement ordering in the source code, and only consider the actual dependences in term of data and control flows.

The complexity of these approaches is usually mostly determined by the complexity of creating the DFG, CFG and/or PDG from the source code.
2.2.4 AST Transformations

As seen in Section 2.1, the AST is the only full representation of the source code. Although the AST is typically augmented with other representations such as the Class Graph or a CFG for doing analyses, the AST remains the main data structure from which the source code can be transformed. In this Section, we review some techniques that are used for transforming source code using the AST. Here we do not focus on specific transformations, but rather on the details of how AST modifications are propagated to the underlying source code.

AST Rewriting

As the AST is a full representation of the code, the simplest way of propagating AST transformations to the source code is just to fully rebuild the source code from the AST after the transformation. This is called rewriting.

Hence, a transformation of the source code can be implemented in three steps (not counting any required analyses):

- Create the AST of the source code using a parser;
- Transform the AST using tree operations;
- Rebuild the source code from the AST (rewriting).

There are some caveats with this method though. Most of the time, a transformation of the source code only affects a small portion of it, and all the rest of the code is left unchanged. To ensure the unaffected code is unmodified using this method, the AST must include nodes for parts of the code that are not relevant for the transformation itself, such as comments and white space. Failure to include comments and white space may result in their removal after the transformation.

Keeping all syntactical details such as white space and comments in the AST would have several disadvantages. It would make the AST bigger, with information that is in most case not relevant. For clone detection for instance, the advantage of the AST is precisely that it abstracts the code from irrelevant information such as white space. For compilation, white space is just useless information.\(^4\)

Origin Tracking

Unlike compilation, when the AST is used to transform source code (such as in a refactoring implementation), it is desirable to leave the unaffected code unmodified. A technique that achieves this goal without explicitly including white space in the AST is origin tracking.

With origin tracking, each node of the AST is augmented with a pair of integers that identifies the location of the underlying code in the source code (represented as text). The two integer values give the start and stop locations in the source code corresponding to the node. This location information can be absent on some nodes for purposes that are discussed subsequently. Yet when the code is parsed, location information is added on all created AST nodes. To allow full reconstruction, the location information includes all irrelevant information from the source code that is present before the node. Hence, white space or comments are indirectly referred by the AST node corresponding to the first relevant construct that follows them. Note that while white space and comments are referred to by AST nodes with such an approach, they are not modeled by individual AST nodes. Hence they are modeled in a non intrusive way.

Then the following single rule has to be followed whenever the AST is transformed:

---

\(^4\)We remind the reader that this thesis focuses on the Java language. For some inferior programming languages such as Python, white space is relevant.
Whenever an AST node is modified (including the addition or removal of a child node), the location information for that node and all its ancestors is removed.

Observe that if a node is moved (i.e. removed from its parent, and attached to a new parent), the location information is removed from both the old and the new parent and all their ancestors. But it is not removed from the moved node itself (unless this node is modified as well).

This rule can be implemented at the level of the classes corresponding to the AST nodes (in the different setter methods). Hence a transformation algorithm does not have to deal explicitly with it.

Then, the following rules are applied in the rewriting process, when the source code is rebuilt from the AST (we assume that the source code is built using a depth-first tree traversal):

- When a node with location information is encountered, the location information is used to quickly copy the corresponding source code (in textual form) from the old version to the version being rebuilt. All the descendants of that node are then skipped, and do not need to be rewritten.

- When a node without location information is encountered, the corresponding source code is regenerated from that node using rewriting, including automatic generation of white space for proper formatting. The code generation then continues recursively on the child nodes, for which the same two rules apply.

In case the AST is not modified, the root node still has location information corresponding to the entire code, which was set during the parsing process. The new source code can hence be generated by copying the entire old source code in a single step, and no recursion is required on the child nodes.

As soon as a single node of the AST is modified, all its ancestors, including the root, have their location information cleared. However, all children of the root except one still have location information, and their underlying source code can be rebuilt by straightforward copies of the old source code. In fact, only the modified node and its ancestors actually have to rewrite the source code. This keeps the needs of rewriting source code to a minimum, and maximizes the amount of source code that is copied in a straightforward way, and is hence left unmodified.

The process is illustrated in Figure 2.9. A single node has been modified and obviously needs to be rewritten (regenerating the underlying code). The same applies to all nodes in the path from the modified
one up to the root. On the other hands, all other children of the rewritten nodes can regenerate the code using origin tracking. As origin tracking regenerates the source code corresponding to the node and all its descendants using simple copy and paste of the original source code, the descendants do not need any processing.

With this approach, all white space and comments are preserved as long as the AST nodes corresponding to the constructs that immediately follows them are not affected by the transformation. Therefore, the white space and comments of the original source code are mostly preserved. Indeed, only white space and comments preceding a node that is deleted or modified will be lost, which is usually relevant in practice: if a node is modified or deleted, it is likely that the comments need to be deleted or modified as well in order to reflect the change.

The Eclipse Java development environment uses a technique essentially based on origin tracking, with small variations and special cases that are beyond the scope of this thesis. A particularity of the version used in Eclipse that is worth mentioning is that it allows child nodes to be added to and removed from a given AST node, but the properties of the AST node itself cannot be modified. To modify a node, one has to remove it from its parent, create a new node, and add it in place of the old one. Because children can be added and removed from a node but the node itself is immutable, the location information only needs to be cleared when children are added or removed. The underlying implementation of maintaining and clearing location information is hence well isolated in the top-level class modeling an abstract AST node. Note that a newly created AST node has obviously no location information.

2.2.5 Domain Specific Languages

The implementation of a complete refactoring can involve many steps: code analyses, data and control flow analyses, and several transformations of the AST and/or of the Class Graph.

As these tasks may be tedious to implement directly using a low level programming language such as Java, there have been attempts to provide higher level languages. These languages aim in being easier to use, but they are specifically targeted to code transformations. As such, they are not general purpose, but domain specific languages.

The use of simplified, domain specific languages for refactoring may also allow the user to script his own refactoring. This is especially relevant as some refactorings are domain specific themselves, and cannot easily be expressed by a composition of “usual” refactorings [45].

**CodeDom** is a library of functions (part of the Microsoft .NET framework) that models code using an Abstract Syntax Tree. It allows parsing, modification and rewriting of the AST. A notable property is that it allows both the C# and Visual Basic languages to use the same AST. In theory it could hence be used to convert from one language to another. In practice, parsing is unimplemented (as of writing this thesis), and the model suffers from several incompletenesses.

**.QL** is a language that allows complex searching, and hence complex analyses of the code to be performed easily [62]. It is based on the SQL query language used in relational database. It allows one to create and execute search criteria using source code specific notions such as method names, classes, overriding, inheritance, references, etc. **.QL** allows the use of several predicates, and permits the construction of recursive queries.

**PROGRES (PROgrammed Gaph REwriting System)** is a language for the implementation of high-level graph transformations [70]. It can be used to perform transformations on the Class Graph for instance.

**JunGL** is a language for the implementation of refactoring at the low level [77]. It provides various predicates (including for flow analyses) and functions to modify the source code. It makes use of the AST for the code transformations, and also uses the Class Graph and the PDG for the analyses.

\[5\text{http://www.sweb.cz/ivan.zderadicka/csparser.html}\]
Stratego/XT is a language specifically targeted to AST transformations [12]. It uses a term rewriting system for describing traversals and transformations. Although Stratego/XT directly operates on the AST, it allows the user to express transformations using the concrete syntax of the underlying language. Hence a user using Stratego/XT mostly works with a familiar syntax.

TXL is another transformation language based on term rewriting [16]. It uses a BNF grammar of the target language, and transformations are expressed as search/replace pairs using a hybrid functional and rule-based language with unification. TXL is not specifically targeted to refactoring, but to the transformation of any structured document whose syntax can be expressed by a BNF grammar.

Velocity is an open-source project directed by the Apache Software Foundation. Velocity is a template engine that includes a language for referencing objects defined in Java code. It is not meant to be used to refactor Java code, but rather to generate Java code from templates, or to dynamically generate HTML documents using templates that refer to Java objects. In the latter case it can be used as a direct replacement of either Java Server Pages (JSP) or javadoc.

While these languages can prove very useful, they are also double-edged swords. They allow one to easily implement what they are intended for, but they may prove to be limited as soon as the task to perform is becoming more complex and beyond their initial purpose. Graph transformations (using PROGRES for instance) are known to be well suited for various modifications of the Class Graph, but poorly adapted to transformations of the AST. Some refactorings such as “extract method” are hence very difficult to implement.

In Chapter 3, we explore a complex refactoring, forming a template method. This complex refactoring somehow illustrates the limitations of domain specific languages, because it requires the use of various general purpose algorithms that have, at first glance, nothing to do with source code transformations.

As an alternative to domain specific languages in general, it has been proposed to use a general purpose language together with high level and complete libraries. A library can be designed in such a way that each construct of a given domain specific language is directly mapped to a library object or method (this has been shown for instance in case of the SQL and HTML languages [46]). Hence, the programming is not significantly more difficult than with a domain specific language, yet it benefits from all the features of a general purpose language. Apart from the advantage of being general purpose, well-known languages such as Java also benefit from wide support in terms of debugging, profiling, refactoring, etc, which is frequently lacking in domain specific languages.

In this thesis the presented algorithms and models have been actually implemented in the form of a Java library rather than on a new or existing domain specific language.

### 2.2.6 Dataflow Languages

The DFG provides valuable information for source code transformations. For instance, the DFG can be used to determine what variables must be passed as arguments and return as results.

The dataflow and control flow graphs also show parts of the code that do not depend on each other, and that can hence be executed in parallel.

Apart from the use of data and control flow graphs for source code analyses and transformations, there have been various proposals of using data and control flow graphs directly as a full featured general purpose programming language [11]. While these languages are named “dataflow languages”, they usually also model control flows. These languages are graph-based, and hence several of them have a straightforward visual representation. Some of them are entirely visual and do not even have an underlying textual syntax.

These dataflow languages benefit from inherent parallelism. Furthermore they allow some complex refactorings such as method extraction to be performed easily. Dataflow languages differ from the data

---

6http://velocity.apache.org
and control flow graph representations discussed in Section 2.1 in that they can be executed. Hence they are not only a representation of a particular aspect of the code; they are the code itself.

Unlike domain specific languages, dataflow languages are not languages whose purpose is to implement an analysis or a refactoring, but rather languages on which analyses or refactorings can be performed easily.

**FOOD**

FOOD is a general-purpose, dataflow language that was developed early during this thesis. It consists of a Class Graph (see Section 2.1.1) and executable graphs. The executable graphs, based on data and control flows, were already presented in Section 2.1.6.

Old dataflow languages and specialized ones are usually only considering flows of data of primitive types (numbers, strings, signals) [11, 32]. In FOOD, executable graphs are not limited to primitive types, but can use arbitrary objects. In terms of object orientation, an executable graph in FOOD is basically the implementation of a method, or its body. FOOD provides full access to the Java API and full support for inheritance, interfaces and mixins.

A basic interpreter as well as an implementation of method extraction have been implemented for the FOOD language [44].

**ProGraph**

ProGraph is another general purpose dataflow language [61]. It has resulted into a commercial product made by Pictorius, featuring an editor, debugger and interpreter. The language has full support for object orientation.

ProGraph is classified in the broader category of visual languages. Note here that a “visual language” is not just a textual language with a visual GUI editor (such as Visual Basic), but a programming language in which the grammar itself consists of visual elements. This distinction is somehow blurred by some visual languages such as Fabrik [40], in which the nodes of a DFG can be either primitives of the language itself, or GUI components.

There is nothing special in ProGraph compared to other dataflow languages. However it was one of the rare dataflow languages that was complete enough to build real-world applications, and that was also commercially successful.

ProGraph is unrelated to refactoring. It is only mentioned here to illustrate that the data flow graph is not only useful to perform source code analyses, but can also be used as the primary syntax of a programming language.

**Soundium**

Soundium is a domain specific dataflow platform targeted to audio, graphics, video and 3D graphics transformations [63]. It has been used to animate visuals during live music performances. Soundium actually consists of three different languages:

- A *processing graph*, which is actually a dataflow language, and is entirely visual. The language supports basic data types as well as domain specific complex data types such as audio and video streams, pictures, and 3D scene elements (based on the OpenGL standard).

- The *sl2* language, a textual language that allows manipulations of the processing graph. This language is primarily used to *build* the processing graph, and hence also to *store* it. Loading a processing graph consists of loading the underlying sl2 code that builds it and executing that code. The language allows both construction and destruction of graph nodes and edges, and can hence
also be used to implement graph transformations. A particular property of the \textit{sl2} language is that instructions are reversible, meaning that an \textit{sl2} program can be executed in backward order. Executing an \textit{sl2} program that builds a graph in backward order will destroy the created graph. Executing an \textit{sl2} program that transforms a graph in backward order will undo the transformation.

- The \textit{design tree}, which is not strictly speaking a language, but a tree structure that allows several processing graphs to be stored in a hierarchical way. Nodes of the tree are modeling the different processing graphs. Edges of the tree are \textit{sl2} programs that transform the processing graph corresponding to the parent node into the processing graph of the child node, and vice versa (as \textit{sl2} is reversible). The root node just corresponds to an empty graph. In practice, only the \textit{sl2} code corresponding to the tree’s edges are stored; the processing graphs do not need to be stored. As a concrete application, the design tree was used to quickly switch from one processing graph to another during a live performance.

In Soundium, operations on the \textit{processing graph} are implemented as refactorings on the underlying \textit{sl2} code. Such operations include node merging, node fusion, node splitting, module extraction, and identifier renaming. Subgraph extraction in particular is comparable to the “extract method” refactoring, but applied to a processing graph. While this refactoring is usually implemented directly on a representation of the code (here it would mean directly on the processing graph), Soundium implements it on the \textit{sl2} language that is actually the code that \textit{generates} the processing graph, and also its only persistent representation. Subgraph extraction in Soundium is hence close to the problem of \textit{program slicing} discussed in Section 2.2.2. Indeed, the part of the processing graph to extract is used as the slicing criterion: exactly the \textit{sl2} instructions that affect it have to be extracted. These instructions are usually not contiguous.

2.2.7 User Interface

The user interface is an important part of any refactoring tool. Unlike design recovery, refactoring is driven by the user. Hence a user interface is necessary to allow the user to supply the inputs of the algorithm. User interface issues will not be addressed directly in this thesis; however this is a large topic on its own. Note that although user interface issues are not discussed, we will still keep the realization of a user interface in mind, and hence point out the inputs of the different algorithms that are supplied by the user.

The different inputs supplied by the user can fall into different categories that correspond to different kinds of user interfaces:

- Simple parameters. Such inputs require a user interface that is relatively easy to build, as it consists of “traditional” user interface components such as text fields, check boxes, etc. Possible simple parameters are:
  - A name for an item (method, field, etc)
  - A selection of one or more existing items
  - Boolean options (such as “replace other occurrences” for an “extract method” refactoring)

- Code selection. This is typically required for the “extract method” refactoring.

- Presentation of the result. In many cases, it is relevant to display a preview of how the refactoring will affect the code, before the refactoring is actually applied, or canceled. Displaying such a preview can be complex for refactorings that affect many files.

- Reporting errors. This is an important feature: there are many cases in which a refactoring cannot be applied, and this can be due to various reasons.
In this thesis, the new presented algorithms will additionally require the following user interface elements:

- Displaying differences between two code fragments (Section 4.2). Such an interface already exists in recent concurrent version systems, in order to compare two versions of the same file.
- Presentation of multiple possible results, and allowing the user to choose the one to apply (Section 6.3). This does not involve major complication compared to the presentation of a unique result.
- Selecting the best matches between two lists of variables (Section 4.3.3).

As discussed in Chapter 2, refactoring algorithms work on abstract representations of the source code. When an error occurs, the error is usually detected using one of the abstract representations. In order to present it to the user, it is necessary to find out the corresponding location in the source code. This is an important issue, which can be solved using origin tracking, as discussed in Section 2.2.4.

2.3 Summary

This chapter summarized the state of the art. In a first part, we showed that the source code needs to be represented by abstract models in order to be analyzed and transformed effectively, and we summarized the main existing representations. In a second part, we discussed the existing analysis and transformation techniques as well as the fields in which they are used.

This study revealed that on one hand, the declarative part of the code can be modeled and manipulated with ease using powerful and well mastered models such as the Class Graph and graph transformations. On the other hand, analyzing and transforming the executable part of the code not only requires tools that are more complex, but also that are mainly targeted to compilation. As compilation no longer deals with the source code once it has been parsed, the underlying tools are inadequate or incomplete for refactoring tasks. As compiling and optimizing is more demanding than refactoring, they also show to be more complex than what would be required for refactoring. This is further developed by a concrete example in the next chapter.
Chapter 3

Case Study: Forming a Template Method

In this chapter, we describe in detail a particular refactoring: forming a template method. This refactoring is used as a case study and main motivation for the new algorithms developed in this thesis. The reason this particular refactoring has been chosen is that it involves several challenges:

- It is linked to various other refactorings, namely rename field, extract method and pull up method. Hence it encompasses several problems linked to statement refactoring and involves a broad category of different problems. This refactoring is also closely related to clone detection and extraction.

- It is complex and reveals the limitations of the existing tools and formalisms. Therefore it requires new algorithms and methodologies.

- It has not yet been implemented in existing development environments.

3.1 Problem Description

The “Form template method” refactoring is best explained step by step using a concrete example. To apply the refactoring, we must be in presence of two methods with similar bodies that belong to two different classes extending the same abstract base class. The refactoring actually applies to the bodies of the two methods in the subclasses.

The refactoring is applicable if the bodies of the two methods are globally similar, with only a few differences. For example, let us assume the two methods have the following bodies:

```java
// In class Rotator
public void rotateAt(Point center, double amount) {
    translate(mult(center, -1 + 5));
    rotate(amount, center);
    translate(center);
    normalize(amount);
}

// In class Skewer
public void skewAt(Point center, double amount) {
    translate(mult(center, -1 - 4));
}
```
These two bodies are very similar. The first line is the same, except that “+ 5” is replaced by “− 4”. The second lines differ, but the last two lines are exactly the same.

A refactoring that naturally comes to mind is to extract the similarities into new methods, as in clone detection and removal. This would result to the following code (we show all the methods one after the other for clarity, yet the methods actually belong to different classes, as specified in the comments. In particular, the new created methods are added to the base class):

```java
// In class Rotator
public void rotateAt(Point center, double amount) {
    tmc(5);
    rotate(amount, center);
    translateAndNormalize();
}

// In class Skewer
public void skewAt(Point center, double amount) {
    tmc(-4);
    amount = skew(amount);
    translateAndNormalize();
}

// In the common base class
protected void tmc(int d) {
    translate(mult(center, -1 + d));
}

protected void translateAndNormalize() {
    translate(center);
    normalize(amount);
}
```

It is clear that in this particular example, which is overwhelmingly simple, there is no real improvement of the code after the refactoring. In practice, clone detection and extraction are only performed in cases where the duplicated code fragments are much bigger and result in significant reduction of the code size.

The idea of the “form template method” refactoring is just the opposite of clone detection and extraction: instead of extracting the duplicated code, we extract all the non-duplicated code. With the same example, this would give:

```java
// In class Rotator
public void rotateAt(Point center, double amount) {
    translate(mult(center, d1()));
    amount = d2(amount, center);
    translate(center);
    normalize(amount);
}
```
protected int d1() {
    return -1 + 5;
}

protected double d2(double amount, Point center) {
    rotate(amount, center);
    return amount;
}

// In class Skewer
public void skewAt(Point center, double amount) {
    translate(mult(center, d1()));
    amount = d2(amount, center);
    translate(center);
    normalize(amount);
}

protected int d1() {
    return -1 - 4;
}

protected double d2(double amount, Point center) {
    return skew(amount);
}

Here, d1 and d2 are the extracted methods. They hold the differences and therefore they have implementations in both classes.

It seems at a first glance that we have just added plenty of new methods and made the whole code much more complicated than it was originally. Yet by looking closely at the result, observe that the methods rotateAt and skewAt now have exactly the same body. Therefore, it is possible to pull up these two methods as a single one in the base class.

The methods d1 and d2 cannot be pulled up as their bodies differ by construction. They must be declared as abstract in the base class so they can still be invoked by the pulled up method. After the refactoring, the following has been added to the base class:

public void templateMethod(Point center, double amount) {
    translate(mult(center, d1()));
    amount = d2(amount, center);
    translate(center);
    normalize(amount);
}

protected abstract int d1();

protected abstract double d2(double amount, Point center);

The two original methods, rotateAt and skewAt, can now be removed from their original classes, and potentially reduce the amount of code. As in any pull up refactoring, all invocations to these two methods in the code must be replaced by invocations to the new templateMethod method.
As its name suggests it, the pulled up method is called the template method\(^1\). This method contains all the similarities in the bodies of the two original methods \texttt{rotateAt} and \texttt{skewAt}. The methods \texttt{d1} and \texttt{d2} are the delegate methods. They are abstract in the base class, and are implemented in both subclasses. They contain all the differences between the two original methods.

Figure 3.1 illustrates the process in the general case from the structural point of view: two classes \texttt{B} and \texttt{C} extend the same class \texttt{A} and contain two methods named \texttt{methodX} and \texttt{methodY} that have similar bodies. After the refactoring, the two methods are pulled up in the base class \texttt{A} as a single method named \texttt{methodXY}. The body of this method contains the statements that are common to both \texttt{methodX} and \texttt{methodY}, and invokes one or more delegate methods (\texttt{delegate1}, \texttt{delegate2}, ...) that actually contain the statements of \texttt{methodX} and \texttt{methodY} that differ. These delegate methods are abstract in the base class \texttt{A} and are implemented in each of the subclasses \texttt{B} and \texttt{C}. They usually have a protected visibility. Observe that the method \texttt{methodXY} is visible from classes \texttt{B} and \texttt{C} through inheritance.

Note that the two subclasses \texttt{B} and \texttt{C} do not need to directly extend the base class \texttt{A}, there can be intermediate classes in the class hierarchy.

Like for the case of clone detection and extraction, no significant improvement can be seen from the previously proposed concrete example, or from the abstract picture of Figure 3.1. But again, the refactoring is usually applied on cases in which the methods have large bodies, with a significant amount of duplicated code. We also remind the reader that in this thesis we will not discuss how to decide whether a given refactoring is relevant or not. Our focus is to perform the refactoring correctly given the user input (even if this does not result in better code).

The “form template method” refactoring can be seen as a generalization of the “pull up method” refactoring for the cases in which the methods’ bodies are not exactly the same (The “pull up” refactoring assumes the two methods to have the same body, at least semantically).

As a summary, the “form template method” is applicable when:

- Two methods of two different classes have the same signature;
- The two classes extend the same abstract class;
- The two classes have similar bodies. However this is not an “absolute” precondition as we are going to cover the problem for arbitrary bodies. In the case the bodies have few or no similarities,

\(^1\)It is clear that a real refactoring tool will usually not just name the created method “templateMethod”, but ask the user to supply a relevant name. The same applies for all the other created methods.
there is just little or no reduction of the amount of code after the refactoring, or in the worst case the refactoring just cannot be done and an error is reported to the user.

And it has to perform the following operations:

- Look for all differences between the two bodies;
- Extract all the differences into new methods in each class. This is actually done by using the “extract method” refactoring multiple times;
- Declare all the extracted methods as abstract in the base class;
- Pull up the remaining two methods as a single method in the base class.

Other definitions

It is necessary to mention that our definition of the “form template method” refactoring is a bit different than the definition given by Martin Fowler in his book [27]. In both definitions the class hierarchy is the same: two methods in two different classes extending the same abstract base class. On the other hand, the way the method bodies are factored is defined slightly differently.

According to the definition of Martin Fowler, the two methods do not need to have similar bodies, but rather to “globally” perform the same steps, although the individual steps can be implemented completely differently in the two methods. With this definition, a template method always looks like a succession of method calls corresponding to the different steps, such as the following:

```java
public void templateMethod() {
    step1();
    step2();
    step3();
    ...
}
```

This would suggest that the template method is entirely made up of calls to delegate methods, which differs from our example in which some of the original code remained. On the other hand, the example given in the book has the following template method:

```java
public double templateMethod() {
    return getBaseAmount() + getTaxAmount();
}
```

In this method, there is still some code that originates from the original two methods, namely the “+” operator, but the template method basically still has the same structure: a sequence of different steps (two in that example).

In this thesis, we do not consider the problem from its structure: we do not assume that the two methods can exactly be decomposed in the same sequence of steps with different implementations. Instead, we focus on the goal: we attempt to remove as much duplicated code as possible by leaving it in the template method.

Therefore, a template method consists in general of both invocations of delegate methods and unmodified code that was present in both original methods. In practice, the result is usually not substantially different. We will also see in the next chapters (namely in Section 6.3.2) that it is sometimes possible to automatically split two methods into similar steps even if their bodies almost completely differ, and hence to also automate the refactoring as defined by Martin Fowler.

50
Other uses

We have assumed a particular class hierarchy as a precondition of the “form template method” refactoring. Namely the presence of the two methods in two different classes extending the same abstract class. We want to point out that the refactoring can also be used in the more general situation where the two methods are just in two arbitrary classes (including the case in which they are in the same class).

In practice, these cases are less relevant, as they require more code to be generated, possibly making the final code much more complicated. The example discussed in the previous section makes use of polymorphic method calls for the delegate methods. If the two methods are not in classes extending the same abstract class, it is no longer possible to invoke the delegate methods in a polymorphic way. There are still two alternative solutions: using the strategy pattern, or using delegates (if supported by the language).

The template method of the example of Section 3.1 would look as follows when using the strategy pattern:

```java
public void templateMethod(Point center, double amount, 
AbstractStrategy s) {
    translate(mult(center, s.d1()));
    amount = s.d2(amount, center);
    translate(center);
    normalize(amount);
}
```

Here, an abstract strategy, s, is passed as argument, and encapsulates the logic of the parts of the method that were different in the two original methods. The AbstractStrategy class declares the two abstract delegate methods d1 and d2. These two methods must be implemented in two different concrete subclasses of the AbstractStrategy. The two concrete subclasses hold the two different implementations corresponding to the two original methods. The subclass corresponding to the rotateAt original method may look as follows:

```java
public class RotatorStrategy extends AbstractStrategy {

    public int d1() {
        return -1 + 5;
    }

    public int d2(double amount, Point center) {
        rotate(amount, center);
        return amount;
    }
}
```

This is not very different than the first example, except that we have explicitly created a hierarchy of three new classes in order to use polymorphic calls. This solution is hence less appealing as the number of new classes it generates may create too much complexity compared to the gain in term of reduction of duplicated code.

Furthermore, the two original methods cannot be removed when a strategy pattern is used, because the correct concrete strategy instance has to be passed explicitly to the template method. The original methods are hence still present, but are reduced to a single line:

```java
public void rotateAt(Point center, double amount) {
```

51
There are still two cases in which it may be relevant to apply the refactoring on methods of arbitrary classes: when the underlying language has built-in support for delegates, and when the refactoring simplifies to a simpler one.

Some programming languages such as Oberon or C# have a built-in support for delegates, or references to methods. With such a language, there is no need to create three new classes to implement a strategy pattern. Instead, references to the actual d1 and d2 methods can be directly passed as arguments to the template method. Again, the two original methods cannot be removed (although they are again reduced to a single line), but no new class needs to be created.

In other cases, it may happen that the differences between the two original methods only consist of constant expressions. Consider for instance the two following methods:

```java
// In first subclass
public void tenPercentRaise(Account c) {
    c.value += c.value * 0.1;
}

// In second subclass
public void fivePercentRaise(Account c) {
    c.value += c.value * 0.05;
}
```

Applying the refactoring blindly would result in the following template method and delegate methods:

```java
// In abstract base class
public void templateMethod(Account c) {
    c.value += c.value * d1();
}

// In first class
public double d1() {
    return 0.1;
}

// In second class
public double d1() {
    return 0.05;
}
```

Yet by noticing that the delegate method d1 only evaluates to a constant expression in both implementations, it is possible to replace it by an argument of the template method:

```java
public void templateMethod(Account c, double amount) {
    c.value += c.value * amount;
}
```

In this case, it is necessary to keep the original methods, because they need to pass the correct value:
// In first subclass
public void tenPercentRaise() {
    templateMethod(0.1);
}

// In second subclass
public void fivePercentRaise() {
    templateMethod(0.05);
}

But in this particular case, we do not need to create new classes to implement a strategy pattern, even if the underlying programming language has no support for delegates. In fact the strategy is actually reduced to constant expressions only, which can be passed directly as arguments\(^2\).

Note that in this particular example, if we then inline all invocations of the two original methods, the “form template method” is equivalent to the “parameterize method” refactoring [28].

In the above example, the parameter of the template method is a constant expression. In theory, the same can be done with expressions involving method calls as long as all the called methods have no side effects. In general, it is not possible to check whether or not a method has side effects with 100% accuracy [50]. In practice though, it is at least sometime possible to determine that a given method has no side effect for sure using the following heuristics:

- A method that has at least one write access to a field has side effects (write accesses to local variables are permitted);
- A method has side effects if it invokes at least one other method that has side effects;
- A method whose source code is not available is considered as having side effects by default. This concerns any methods located in a library whose source code is not available, but also any native method (whose implementation is written in another language).

A consequence of the last point is that access to the source code of libraries used by the application can greatly increase the number of methods free from side effects that are recognized. In particular, many methods from the standard Java API have no side effects.

While several functions without side effects might still be missed with such an heuristic, this scheme can at least detect all “standard” getter methods (i.e. those that just return a field) as being free from side effects. In practice this may already cover a substantial number of relevant situations.

Context

In the context of refactoring tools, the proposed case study encourages research toward more complex refactorings, compared to what has already been implemented. This is also the main goal of this thesis. Our aim is not to focus on formalization, generalization, abstraction, implementation or modeling of existing techniques. It is also not our goal to focus on abstract theories and models. Instead we aim to provide the basis for a working implementation of the “form template method” on the Java language.

The availability of more complex refactorings (and in general the availability of new tools that help in the development of applications) may seem unnecessary at first glance. For instance, one may ask oneself if forming a template method is really useful in concrete projects. However, exploration of existing

---

2As of writing this thesis, there are plans to introduce closures in the new version of Java (version 7). Closures are halfway between constant expressions and delegates: they are blocks of code that can be passed as argument for lazy evaluation, similar to anonymous or in-line delegates. Closures would hence provide another alternative to the strategy pattern that covers more cases than the use of constant expressions.
tools and their history showed that the introduction of new techniques has also changed the behavior of programmers and frequently resulted in the new technique to be increasingly used over time [36, 37]. For instance, before automated refactoring existed, programmers used to think very carefully about their design, including details like the name they would give to a class. Now that an automated and reliable “rename class” refactoring exists in almost all development environments, programmers tend to just give some potentially good name without spending too much time, and just change it (using refactoring) if it later turns out to be inappropriate.

Hence, although the “form template method” refactoring does not seem as helpful as renaming or method extraction, it may prove to be increasingly useful once it is implemented. In the meanwhile, this refactoring has the property of involving various sub problems, and hence the proposed solution applies to a broad class of problems. This is highlighted in the next section.

3.2 Challenges

In the previous section we have defined a complex refactoring, forming a template method. The objective of the remaining chapters of this thesis is to develop an algorithm that performs such a refactoring automatically. In this section, we first give some intuitive highlights on the implied challenges and on the solutions we are going to propose.

3.2.1 Code Differentiation

As a first step of the proposed refactoring, it is necessary to compare two method bodies, and to figure out what fragments are the same and what fragments are different. While this looks quite similar to clone detection, there are three major differences though.

Ordering

Consider for example the following two (artificial) simple methods:

```java
public void method1() {
    a1(); a2();
    b1(); b2();
    c1(); c2();
    d1(); d2();
}
```

```java
public void method2() {
    a1(); a2();
    c1(); c2();
    b1(); b2();
    d1(); d2();
}
```

They are similar, with the exception of the order of the occurrences of the “b1(); b2();” and “c1(); c2();” lines. Obviously it would be useful if we can prove that the two lines are independent statements; which can be done using techniques discussed in Section 2.2.2 for instance.

However, in the general case we cannot assume that arbitrary statements can be reordered. Assuming that the two statements cannot be reordered, a clone detection and extraction tool may at best find four different clones corresponding to the four lines of code. Although this is worse than a single clone for all
lines of code (if statement reordering is possible), the tool could still reduce the amount of code in every line.

In the case of forming a template method, we want to extract only the differences, and to leave the similarities in the template method. If the two statements could be reordered, there would be nothing to extract and the template method would contain all the four lines of code. In the general case though, we cannot leave every line in the template method if they are not in the same order.

Obviously, the first and last lines can be left in the template method. But if we choose to leave the second line of the first method in the template, we have to leave the corresponding third line of the second method, and vice-versa. In both cases, we are forced to extract at least one line of each method, or else the original ordering of the statements cannot be respected. In fact there are three different solutions for the template method:

```java
public void templateMethod1() {
    a1(); a2();
    delegate1();
    d1(); d2();
}

public void templateMethod2() {
    a1(); a2();
    delegate1();
    b1(); b2();
    delegate2();
    d1(); d2();
}

public void templateMethod3() {
    a1(); a2();
    delegate1();
    c1(); c2();
    delegate2();
    d1(); d2();
}
```

In the first solution, the `delegate1` method is implemented with the second and third lines, once in one order and once in the other order\(^3\). In the second solution, the `delegate1` method has one empty implementation, and one implementation with “c1(); c2();”, whereas the `delegate2` has one implementation with c1(); c2(); and one empty implementation. The third case is similar to the second one, except that “c1(); c2();” is left in the template instead of “b1(); b2();”.

The first thing to note is that none of these solutions is optimal: in all cases we have extracted in delegate methods code fragments that still contain some duplication. The first solution looks the most appealing as it only creates a single delegate method. The second and third solutions create two delegate methods, and each one has one of the two implementations that is empty, which is not very elegant. However, the second and third solutions leave more similar statements in the template method compared to the first one (three original statements instead of two).

In other words, an algorithm that maximizes the number of similarities that are left in the template method is not necessary the optimal solution. However, no solution fully removes all duplications: either

---

\(^3\)Recall that we assume two methods of two different classes extending the same base class. Hence a delegate method always has two different implementations, one in each class.
“b1(); b2();” or “c1(); c2()” (or both in the first solution) are then duplicated in two delegate methods.

**Variable Matching**

As with any refactoring, it is more appealing if the algorithm can recognize similarities in the semantics although they are not directly visible in the syntax. One desirable such feature, which is present in most clone detection tools, is to be able to match variables that are the same even if they have different names.

Consider the following two methods:

```java
public void method1(int z) {
    int x = z * 2;
    print(x);
    store(x);
    int y = x - 4;
    return y;
}

public void method2(int z) {
    int y = z * 2
    print(y);
    store(y);
    y++;
    return y;
}
```

If we assume that only variables that have the same name are equivalent, we can only match the last line and the “z * 2” expression between the two methods. On the other hand, if we assume that the roles of x and y are reversed in the second method, we can match all the three first lines, but not the last one. The latter choice gives the highest number of similar statements and is hence a better choice than the former. Yet finding the optimal matching between the variables of the two methods is not as simple as it seems at first glance:

- In order to check whether a given statement is the same or not between the two methods, we need to know how the variables are matched between the two methods. Indeed, the first statements in the above example are the same if and only if x is matched with y.
- But to know what variables are matched, we need to know what statements are the same in both methods. Indeed, only variables that appear in the same statements are likely to be the same.

This problem as well as a feasible solution are discussed later in Section 4.3.

**Exact Matching**

When forming a template method, it is necessary to first compare two method bodies. As the next step is to extract the fragments that are different, it is necessary to make an exact comparison. Here “exact” mean 100% precision, but not necessarily 100% recall.

This differs from clone detection techniques which are usually based on approximate comparisons. While several clone detection tools are then using a second, exact algorithm to refine the result (and to filter out false positives), this second algorithm has a much simpler task to perform: it does not have to look for clones, but just to check whether two detected clones are really equal or not. Eventually, the
algorithm can do some minor changes to the clones for instance if they only differ by statements that can
be safely removed from them.

The reason clone detection tools use approximate comparisons in a first step is mainly a performance
concern. Clone detection is usually used on large projects, and needs to give results within a reasonable
amount of time. In the case of refactoring, only small portions of the code are involved. Hence perfor-
man c e c onsiderations are less important. Another reason for using approximate comparisons (less than
100% precision) in clone detection is that it allows the use of simple and fast heuristics that may further
boost the recall [23].

When forming a template method, the two code fragments (the body of the two methods) are supplied
as an input, and the similarities and differences must be detected with 100% precision. While it would
be possible to start with an approximate comparison and to refine the result with an exact comparison,
we propose in Section 4.1 a new technique that allows an exact comparison right from the start. The
technique furthermore runs in linear time, making it appealing as well for clone detection.

3.2.2 How to Deal With the “Extract Method” Preconditions

A second challenge is related to method extraction: in order to build a template method, it is necessary to
take all the code that differs from the two original bodies. These code extractions are applications of
the “extract method” refactoring. As a refactoring, method extraction works like any other refactoring: it
has inputs and preconditions. If the preconditions are not fulfilled, the refactoring cannot be performed
and a refactoring tool will just supply a descriptive error message to the user.

A template method can only be formed if only equivalent code is left in the two original methods. A
consequence is that if only one method extraction cannot be performed, the template cannot be formed
and the entire refactoring is impossible. Therefore, Chapter 6 is entirely devoted to small transformations
that allow a method to be extracted with small modifications whenever it cannot be extracted “as is”.

Comparable transformations have been proposed in the field of clone extraction, when a detected
clone cannot be extracted “as is”. While the same techniques can be used when forming a template
method, we will propose other and new techniques. While the goal is the same, the way they operate is
more adapted to refactoring.

Clone detection and extraction is usually considered as design recovery rather than refactoring. A
consequence is that the process is less controlled by the user. Consequently, many clone extraction tools
may perform complex modifications of the code to allow a clone to be extracted. In the context of
refactoring, most existing tools prefer reporting errors rather than automatically transforming the code
when something cannot be done in a straightforward way [30].

An example is the extraction of a method that modifies two local variables. Such a method would
need to return two values, which cannot be expressed in a straightforward way in Java. In the context of
copy extraction, it may be relevant to play some tricks such as returning an array, or an instance of a new
class with two fields holding the results. In the context of refactoring though, these tricks might be viewed
as inelegant by the user or just not suited to the application’s design. While several possibilities could be
proposed to the user of a refactoring tool (including aborting the transformation), we will propose some
new transformations schemes that are more natural, and can sometimes allow the extraction of difficult
methods with only minor modifications of the code.

3.2.3 Reversing the Clone Extraction Process

The process of clone extraction extracts identical code fragments by definition, while the process of
forming a template method has to extract code fragments that differ. While the two problems seem to be
related, the extraction of code fragments that differ involves some additional challenges:
• Extracting different code fragments means that potentially different variables are accessed by each fragment. Hence the arguments and results of the extracted methods must consider the data flow requirements of the two code fragments at the same time. The reason is that while an extracted method has two different implementations, it must have a unique signature so that it can be invoked in a polymorphic way by the template method.

• In addition to accessing different variables (i.e. having different data flow), the two code fragments might also have different control flows, such as one of the two fragments having a `break` statement, but not the other.

• Differences between the two methods may also include “degenerated” cases in which some code is only present in one of the two original methods. This situation has to be coped with as well.

• Finally, in complex situations two code fragments to extract might occur in different scopes. One of them for instance could be within a loop but not the other.

Chapter 6 discusses various techniques to address these issues, mainly by merging information (such as data and control flows) from the two code fragments.

### 3.2.4 Performance Considerations

The expression “identical code” has been mentioned many times. We recall that checking for exact semantic equivalence is not decidable [81]. Hence “identical code” is only defined by what an actual tool is able to identify as identical. The only restriction in our case is that the precision must be of 100% (no false positives), while there is no restriction on the recall (true negatives). In practice though, it is obviously desirable to have a recall as high as possible.

There are several ways to improve the recall when comparing code for duplications or differences. Most of them are based on semantic information. Some of them can be expensive in term of computation, especially those based on the control flow graph (CFG) or program dependence graph (PDG), mainly because of the computational costs of creating these graphs. Indeed, the construction of such a graph is substantially slower than, for instance, the construction of the abstract syntax tree (AST).

Performance is important for a refactoring tool, as refactoring is usually done in “real-time”, that is, intermixed with the user’s activities of writing code, testing and debugging. Performance is less important in the case of clone detection as the user usually only interacts after the clone detection process has finished. In the case of compilation (which also uses similar techniques for optimization), such techniques are usually only used in an “optimized” mode when an application’s release is built, but not during the normal development cycle in which optimizations are usually deactivated. Hence it is acceptable to use potentially slow techniques as long as they can be deactivated during the development cycles.

Chapter 5 is devoted to a new technique that performs a data and control flow analysis using the AST directly, and hence significantly faster than existing versions based on the CFG or PDG4.

### 3.2.5 Other Challenges

There are several other challenges that are involved in the process of forming a template method. Most of them have already been addressed in the context of other refactorings and are hence not developed in this thesis.

• Parsing source code into an AST. This is done by any compiler and is a well mastered process. Implementations of the algorithms presented in this thesis are using the AST parser of the Eclipse JDT (Java Development Tools).

4Recall that while data and control flow dependences are trivial to infer from the CFG or PDG, the construction of these two graphs has to be considered as well, and is significantly slower than the construction of the AST.
• Building the Class Graph (which basically consists in resolving all the bindings). This is again done by any compiler. Implementations of the algorithms in this thesis also use the Eclipse JDT. Note that while this tool does not create a graph in a formal way (using explicit classes to model nodes and edges), the Class Graph is indirectly formed by regular Java objects (representing binding information) and references between them. The Java objects are the nodes of the graph and the references implicitly model the edges of the graph.

• Synchronizing the source code as the AST is modified and as new nodes are added. This is done using “origin tracking” as discussed in Section 2.2.4 and is supported by several existing tools.

• Checking name conflicts and renaming items. When delegate methods are created they must be given a name (either a synthetic name, or preferably a name supplied by the user). In both cases it is necessary to check that no other method has the same name in the same scope, or else compiler errors may result. This is already well addressed in the literature on the “rename field” and “rename method” refactorings [28, 77].

• Checking for the visibilities of the accessed fields. After the template method is pulled up in the base class, some accessed fields may no longer be visible. This can be done quite easily with the Class Graph [58], including the handling of modifiers such as public, private, etc.

• The “pull up method” refactoring is the last step to perform when forming a template method. This refactoring is well mastered using the Class Graph and the AST.

• The implementation of the user interface of a refactoring tool using the proposed algorithms will not be discussed in depth, although we will point out the main properties and features such a tool should offer in term of user interactions.

Hence we will address the following tasks, for which we will summarize previous techniques (when existing), address their limitations, and propose new ones:

• Looking for code duplications;
• Comparing two method bodies for differences and similarities;
• Recognizing variables with different names but identical semantics within the above process;
• Performing a fast data and control flow analysis (without requiring the construction of a DFG or CFG);
• Modifying code fragments that cannot be extracted “as is” in a method, in such a way they can be extracted safely.

3.3 Summary and Approach

Summary

We illustrated a complex refactoring, and we highlighted all the new challenges that an automated or semi-automated implementation would involve. This case study also showed the limitations of the existing techniques, and we introduced the new methodologies we are about to propose.
Our Approach

As already discussed, the problem of forming a template method shares similarities with the problem of clone extraction. In the literature, clone extraction is typically presented as a two-step process:

1. Detect the clones
2. Extract the clones

While the same approach could also be used to form a template method, we propose a more fine-grained approach that consists of four different steps:

1. Detect differences and similarities between the two methods
2. Check the preconditions of every required method extraction, and prepare its signature
3. Attempt to resolve unsatisfied preconditions
4. Perform the method extraction and pull up the template method

In these four steps, Steps 2 to 4 correspond to Step 2 of the clone extraction process. The next three chapters are presenting improvements and new algorithms for the first three steps of the proposed approach: Chapter 4 deals with Step 1, detecting differences and similarities between two methods. Chapter 5 propose a new approach for Step 2; more precisely, it introduces a new algorithm for analyzing the data and control flows of the method to extract, and explain how to use them in the process of preparing method extraction. Chapter 6 finally discusses various ways in which unsatisfied preconditions can be resolved automatically or semi-automatically.

Note that the last step is not addressed in this thesis, as we entirely rely on existing techniques for that part of the process. The necessary techniques are already implemented by the JDT (Java Development Tools) of the Eclipse project for instance, and were briefly presented in Sections 2.2.1 and 2.2.4.
Chapter 4

New Approaches to Statement Comparison

This chapter introduces new approaches for the problem of comparing code. Section 4.1 presents the approach in the context of clone detection. Section 4.2 proposes modifications of the proposed method to make it suitable to the problem of comparing statements when forming a template method. Section 4.3 discusses a possible way of recognizing equivalent variables that have different names.

Our target is to propose a code comparison algorithm that meets the following requirements, all of which are focused to the “form template method” refactoring (as opposed to a clone detection tool used for design recovery):

- Code comparison with 100% precision. Unlike clone detection, found duplicated statements have to match exactly. Failure to provide 100% precision would have the consequence that not all differences can be extracted in delegate methods. As a result the whole refactoring could not be performed. Note that 100% recall is not required, although high recall is desirable.

- Syntax abstraction. A refactoring tool is expected to know about the syntax of the language and to appropriately ignore irrelevant stuff like blank lines and white space. It should also differentiate variables with the same name having different scopes.

- Near-linear ($O(n \log(n))$) complexity. As refactoring is a “real-time” task (i.e. intermixed with code writing, testing and debugging), slow and iterative algorithms are not adequate. Currently only string-based and token-based approaches can achieve this complexity; but they do not allow syntax abstraction without introducing false positives. With respect to the previous points, it is clear that a new technique is required.

- Simple implementation. Our position is to not only consider the quality of the algorithms based on their effectiveness, but also on their simplicity and on how easy they are to implement. This is consistent with Agile methodologies that takes as granted the fact that technology tends to evolve more rapidly than it is possible to update legacy code.

4.1 Clone Detection using AST Traversals

This section introduces a new clone detection technique based on the AST. This technique assumes that both the AST and the Class Graph are available (actually the bindings of the variables). Their construction is part of any compiler and is hence not discussed. They can both be done in linear time [73].

61
The proposed approach to clone detection is decomposed into the following steps:

- Parse the source code into an AST and resolve the bindings;
- Create a list view of the AST using a post-order traversal;
- Look for clones in the list view using the LZ77 algorithm;
- Correct artifacts.

Each step is detailed in the following sections. We omit the very first step that is part of any compiler and is a well mastered process.

### 4.1.1 The Traversal List

This step consists in creating a list view of the AST. This is done by performing a traversal of the AST. Although we will only consider post-order traversals in the remainder of this chapter, the same results are also obtained using in-order or pre-order traversals, with only minor differences.

Consider for instance the following source code:

```plaintext
save(get(x * 3) - (x + 6) - 4);
print(get(x * 3) - (x * 5) + 4);
```

Figure 4.1 shows the corresponding AST. By performing a post-order traversal of the AST, one obtains the following token list (where each token is a node of the AST):

```
[x, 3, *, get, x, 6, +, -, 4, -, save, x, 3, *, get, x, 5, *, -, 4, +, print, block]
```

The proposed approach answers to the above mentioned requirements by providing 100% precision, syntax abstraction, $O(n \log(n))$ complexity and simplicity. 100% precision is achieved by the use of exact comparisons instead of heuristics; syntax abstraction is provided by the use of the AST; near-linear complexity is achieved by the clever use of simple algorithms and data structures; and simplicity is achieved by mostly relying on existing algorithms and frameworks.
There is an important feature of the proposed algorithm to understand at this point: the original AST is not destroyed, the created list just links the existing nodes of the AST in a “post-order traversal” order (for example using a singly or doubly linked list data structure). In other words, tokens of the list are just nodes of the AST. Each node hence contains the following structural information after this step:

- The parent node in the AST;
- The child nodes in the AST, ordered;
- The previous node in the “post-order traversal” order;
- The next node in the “post-order traversal” order.

From the point of view of the implementation, the two structures exist separately: the AST structure is the existing one provided by the Eclipse JDT, and the list structure is just an `ArrayList` instance (from the `java.util` package). However, each AST node exists only once, and the two structures (AST and list) are hence referring to the same node instances.

This reference aliasing is important, as it allows us to benefit from the advantages of both representations when analyzing the code:

- The list is a simple data structure, and permits the use of a wide range of efficient algorithms.
- The AST is more complex, yet it provides all the structural information that is lost to a large extent by the list.

There is another crucial point to note: the variables such as “x” are compared not only using the AST node (which only holds the name), but also using the corresponding binding (the corresponding node in the Class Graph). This is mandatory in order to properly differentiate two variables with the same name but different scopes.

Before introducing the second step, let us review the produced token list:

\[x, 3, *, get, x, 6, +, -, 4, -, save, x, 3, *, get, x, 5, *, -, 4, +, print, block\]

Looking at this list, code duplications are clearly identifiable as duplications of sublists. For instance, the duplicated expression “get(x * 3)” in the source code shows up as the duplicated sublist \[x, 3, *, get\] in the token list. However, if we search for the longest duplicated sublists, we find that not only the sublist \[x, 3, *, get\] is duplicated, but the sublist \[x, 3, *, get, x\]. Unfortunately, the extra “x” token is not part of the cloned expression, and does not really correspond to any clone at all. This extra token is an artifact of the fact that the list view of the code has lost most of the structural information present in the AST. Looking at the AST, it is clear that the extra “x” is part of another expression. It is only following the duplicated tokens in the token list by accident. The remainder of the algorithm hence consists of two separate steps:

- Look for code duplications by detecting duplicated sublists in the token list. As suggested, this will produce some artifacts because the token list itself does not exhibit the entire AST structure.
- Remove and correct the artifacts of the previous step. This is done by looking back at the AST structure. This step relies on the fact that both structures (the list and the AST) use reference aliasing: each token of the list is also connected to the AST and can therefore use the structural information present in the AST.
4.1.2 The LZ77 Algorithm

The second step looks for duplicated sublists in the token list. The list data structure is a very simple one and hence there are many fast algorithms available. A candidate is the LZ77 algorithm [64], which works in $O(n)$ average complexity. This algorithm is unusual in the area of refactoring, and comes from the area of data compression and coding [13]. It is indeed a main part of the deflate algorithm that is used in the popular zip compression utility.

Note the strong relationship between data compression utilities and clone detection: in both cases we want to reduce the amount of duplication. In the case of data compression, the final goal is to reduce the file size, whereas the goal of clone detection and extraction is to improve existing design. However, improved design means a reduction in the duplication (and thus of the code size) in the case of clone extraction.

We first describe the LZ77 algorithm in general, and then show how it can be applied to the second step of the presented clone detection algorithm.

Definition and Use

The LZ77 algorithm has been proposed as a way of removing duplications in arbitrary byte streams (in practice the byte streams are files) [64]. However, it does not depend on the nature of the elements of the stream, and only relies on some properties regarding the elements’ type. These properties are the following:

- The elements must be comparable.
- A hash function must be defined for groups of two or three elements. This is required for efficient implementations, but not for a straightforward implementation, as discussed in the next section.
- The element type must be extensible to accept a new special element to encode a “match” (see later).
- There must exist an injective relation between the $[1, N]$ interval of integer values and a group of a small number of elements (typically between 1 and 4). The value of $N$ greatly varies from one implementation to another; a typical value is between 1000 and 100000. A power of two such as 65536 (which equals $2^{16}$) is frequently used as it is optimally encoded by two bytes.

The LZ77 works by iterating over the elements of the stream or list, and maintaining a sliding window. The sliding window constantly maintains the last $N$ elements that have been encountered. The sliding window is typically filled with a default value at the beginning of the algorithm.

As the elements of the list are visited, the algorithm constantly looks for a sequence of two or more visited elements occurring in the sliding window. If this is the case, it means that the sequence is duplicated. In that case, the algorithm replaces the duplicated sequence by a “reference” to the previous occurrence. The reference is formed by a triplet consisting of:

- A special “match” element. To add such an element, the element type must be extensible to allow it.
- A number that corresponds to the distance between the duplicated sequence, and its previous occurrence in the sliding window. If the sliding window has a size of $N$ elements, the distance can vary from 1 to $N$. To add this number to the stream or list, the injective relation discussed above is used.
- A number that corresponds to the length of the duplicated sequence.
With the following token list (from the previous section):

\[ x, 3, *, get, x, 6, +, -, 4, -, save, x, 3, *, get, x, 5, *
\]
\[-4, +, print, block \]

The LZ77 algorithm would output the following new token list:

\[ x, 3, *, get, x, 6, +, -, 4, -, save, \{M\}, \{11\}, \{5\}, 5,
\]
\[-4, +, print, block \]

Here, \(\{M\}\) denotes the special “match” element, while \(\{11\}\) and \(\{5\}\) correspond to the distance and length values, encoded using the underlying element type. Note that the length of the new list is shorter by two tokens, because the duplicated sublist of five elements has been replaced by a “reference” to the previous occurrence, taking only three elements. The number of elements required to encode a reference can vary from one implementation to another depending on the size \(N\) of the sliding window and the cardinality of the element’s type. In some cases for instance it might be necessary to use more than one element to encode an integer value, whereas in other cases the two values (distance and length) can eventually be combined in a single element. In all cases, duplicated subsequences are only replaced by a reference if the reference can be encoded by fewer elements than the length of the duplication. In all other cases, this would increase the size of the list and would hence not be adequate in the context of data compression. In the above example for instance, the \([-4]\) sublist is duplicated; however, it has not been replaced by a reference as this would require three tokens and hence would increase the size of the list.

**Implementation**

A straightforward implementation of the LZ77 algorithm requires \(N\) comparisons for every visited element of the list, because every visited element has to be compared against the \(N\) elements of the sliding window for a potential match.

As soon as one or more potential matches are found (by potential, we mean that only the first element of a possibly duplicated sequence has been matched yet), the algorithm has to “track” these matches as the next elements are visited to figure out the length of the duplicated sequence: as long as the next elements are also matched, the length of the duplicated sequence is increased. As soon as an element no longer matches, the duplicated sequence can be replaced by a reference if its length is long enough.

However, even when potential matches are tracked, it is necessary to continue looking for all other potential new matches, as illustrated by the following example:

\[ a \ b \ u \ b \ u \ x \ v \ a \ b \ c \ a \ b \ u \ x \ v \]

At the time the last “a” is visited, two potential matches are found, corresponding to the two previous occurrences of the element “a”. These two matches continue to be tracked as the next element, “b”, is visited. However, failure to also look for all other possible new matches in the sliding window with the “b” element would result in missing the longest duplicated sequence of the list, namely “[b u x v]”, and only finding the shorter sequence “[a b u]” instead.

There are many other subtle improvements that can be introduced in the LZ77 algorithm that are beyond the scope of this thesis. The reader can refer to papers on the topic [13, 64].

Looking for every \(N\) elements of the sliding window for every visited element is not very effective in term of performance. If \(S\) is the length of the entire list, the complexity of the straightforward algorithm is \(O(NS)\).

In practice, a hash table is used in addition to the sliding window to improve performance. The hash table contains the hash codes of all consecutive groups of \(M\) elements of the sliding window. There are \(N - M + 1\) such groups. \(M\) is the minimal length of a duplicated sequence that is worth replacing by a
reference, 4 in our example. To look for potential duplications when a new element is encountered, the hash code of the new element and the $M - 1$ previous elements is computed, and is looked for in the hash table. Only if there is a match in the hash table, is the match verified using the sliding window\(^1\). In case the match is verified, it actually starts being tracked.

With a good hash function, looking for potential matches in the sliding window is reduced to $O(1)$ average complexity (the cost of a hash table lookup). However, many potential matches can be tracked at the same time. However, the complexity of tracking multiple matches can also be reduced to achieve a global $O(N)$ complexity for the algorithm using suffix trees [13].

### Application to Clone Detection

We propose to use the LZ77 algorithm on the token list described in Section 4.1.1 for clone detection. However, the LZ77 was originally intended for data compression. To apply it to clone detection, it requires some modifications:

- The duplicated sequences are recorded in a separate structure, but the token list itself is not modified. The reason is that the actual coding of duplications using references will be replaced by the clone extraction process. As such, the restrictions that the token element type must accept a new “match” element and must allow coding of integer values are no longer needed.

- Some optimizations of the LZ77 that are proposed in the literature must be disregarded. One example is an optimization that allows duplicated subsequences to overlap [64], which makes no real sense for clone extraction.

Another question is the choice of the sliding window size $N$. Smaller values usually result in smaller memory usage; however duplications that are more than $N$ tokens apart cannot be detected. In the context of data compression, there are several reasons to keep $N$ relatively small (compared to the file size). Large values require more memory for the decompression. Large values also require more elements to encode distances and lengths, leading to a reduction of the effectiveness in term of compression.

In the context of clone detection, it is relevant to just leave the value of $N$ unbounded. In other words, the hash table constantly grows as the code is traversed; and there is no need to explicitly maintain a sliding window because the sliding window is implicitly given at any time by the interval from the beginning of the token list up to the currently visited token.

### 4.1.3 Correcting Artifacts

We now continue with the next and last step of the clone detection algorithm: correcting artifacts.

Recall that starting from the initial code example:

```plaintext
save(get(x * 3) - (x + 6) - 4);
print(get(x * 3) - (x * 5) + 4);
```

The AST traversal gives the following token list:

```plaintext
[x, 3, *, get, x, 6, +, -, 4, -, save, x, 3, *, get, x, 5, *
*-, 4, +, print, block]
```

And the LZ77 algorithm identifies that the “$[x, 3, *, get, x]” sublist (among other) is duplicated. Unfortunately, this token list does not correspond to a single expression. The first four tokens, “$[x, 3, *, get]” correspond to the “get(x * 3)” expression whereas the last “x” token is the

\(^1\)Hash code comparison alone can yield false positives.
occurrence of the \( x \) variable in the \( (x + 6) \) expression. It is obviously hardly possible to extract a clone that consists of two different expressions.

The problem arises from the fact that the LZ77 algorithm works on a token list, and hence does not take any regard to the original tree structure: the list contains less information than the AST. As a result, many such “artifacts” are expected in the output.

Rather than abandoning the LZ77 algorithm, it is possible to correct these artifacts afterwards. Here it is crucial that every token of the list is also a node of the AST, and hence knows about its parent node and child nodes. Indeed, artifacts can only be corrected with the help of the structural information available in the original AST.

We propose two options for correcting artifacts: completing expressions and splitting expressions. The first option, completing expressions, is conservative in that it only adds tokens to the detected duplicated sublists. The second option, splitting expressions, only removes tokens from the detected duplicated sublists. It may also split the detected sublists into smaller ones.

When used for clone extraction, the first option is not appealing, because adding tokens potentially means adding elements that are not duplicated, and that may prevent the clone extraction. It is still of interest when forming a template method, as discussed later in Section 4.2.3.

**Completing Expressions**

The first proposed solution to correct the artifacts of the LZ77 algorithm is to attempt to complete the detected expressions so that they each cover a single, but complete expression.

We first give a few definitions related to nodes of an AST:

- **Ancestor** \( A \) of a given node \( N \) is any node that is reachable from \( A \) by following the “parent” reference zero or more times. This can also be defined recursively as follows:
  - Any node \( N \) is an ancestor of itself
  - If \( A \) is an ancestor of \( N \), then \( A\.parent \) is an ancestor of \( N \)

  Basically, all nodes that form the path from the root node to the node \( N \) are ancestors of \( N \). The root node is hence the ancestor of every node of the tree.

- A node \( D \) is a descendant of a given node \( N \) if \( N \) is an ancestor of \( D \).

- A common ancestor of a set of nodes \( N_0, N_1, N_2, ... \) is a node \( A \) such that \( A \) is an ancestor of every node \( N_i \). As the root node is the ancestor of every node, at least one common ancestor exists (the root node) for any non-empty set of nodes.

- The lowest common ancestor of the nodes \( N_0, N_1, N_2, ... \) is the common ancestor \( A \) such that no children of \( A \) is a common ancestor of all the nodes \( N_i \).

Completing an expression is done using the AST structure, in the following steps:

- Look for the lowest common ancestor of all the tokens of the detected duplicated sublist.

  Consider the subtree formed by the lowest common ancestor as the actual expression. In other words, the corresponding tokens can be obtained by adding all the missing descendants of the lowest common ancestor.

Figure 4.2 illustrates the process for the following detected duplicated sublist of tokens:

\[ [x, 3, *, get, x] \]
Figure 4.2: The process of completing a detected cloned expression (illustrated on the first occurrence). The result is a complete subtree of the AST.

The results are obviously different depending on which occurrence of the clone we are considering, because we are adding tokens that have not been matched. The two results are the following lists of tokens:

\[
\begin{align*}
[x, \ 3, \ *, \ get, \ x, \ 6, \ +, \ -] \\
[x, \ 3, \ *, \ get, \ x, \ 5, \ *, \ -]
\end{align*}
\]

They correspond to the following two expressions:

\[
\begin{align*}
get(x \times 3) - (x + 6) \\
get(x \times 3) - (x \times 5)
\end{align*}
\]

Each of these two expressions can now be safely extracted if desired. As previously suggested, the result is not appealing for clone detection because unmatched statements have been added. As a result, clone extraction can get difficult or impossible. In the above case, clone extraction remains possible, but the extracted method must accept an argument to specify the “varying” part. The extracted method may look as follows:

```java
double extracted(double x, double value) {
    return get(x * 3) - value;
}
```

And the original code can then be reduced to:

```java
save(extracted(x, x + 6) - 4);
print(extracted(x, x * 5) + 4);
```

**Splitting Expressions**

Completing expressions is not appealing for clone extraction, as it adds tokens that are not matched. An alternative solution is to split the detected token sublists. This can be done by ordering the tokens from their depth in the AST (their distance from the root node), and then iterating on the ordered tokens. For each visited token:
• If all its descendants belong to the sublist, this token and all its descendants are put in a new sublist. The descendants are skipped from the next iterations.

• If some of its descendants are missing in the sublist, this token is removed from the sublist. The iteration continues normally.

Only the newly created sublists are then considered.

Using our example:

\[ [x, 3, *, \text{get}, \ x] \]

The get token is visited first because it is the closest to the root. All its descendants are present in the sublist, and hence they are all put in a new sublist: “\([x, 3, *, \text{get}]\)”. Note that when forming a new sublist, the order of the tokens in the original sublist is preserved. Skipping all descendants, only the last x token is then visited. This token has no descendants apart from itself. Hence it is put alone in a second new sublist: “[x]”.

The result of splitting expressions is that a single duplicated sublist will result in one or more sublists. In our case, we end up with two sublists, corresponding to two different clones: “\(\text{get}(x \times 3)\)” and “x”. It is clear that in practice, the second clone (in this particular example, “x”) is too short to be worth considered, and will be filtered out after this step.

A last step is to check whether some of the created sublists appear to match. Consider for instance the following code:

\[
\begin{align*}
\text{save}(\text{get}(x \times 3) - (\text{get}(x \times 3) + 6) - 4); \\
\text{print}(\text{get}(x \times 3) - (\text{get}(x \times 3) \times 5) + 4);
\end{align*}
\]

The LZ77 algorithm would find the duplicated sublist “\([x, 3, *, \text{get}, x, 3, *, \text{get}]\)”. By splitting expressions, we would end with two instances of the following sublist: “\([x, 3, *, \text{get}]\)”. These two instances can be merged into as single clone that occurs four times: the “\(\text{get}(x \times 3)\)” expression.

Handling Block Nodes

When splitting or completing expressions, we assumed that only a complete AST subtree can be safely extracted. This assumption can be relaxed in some cases, in the presence of a block node. A block node is an AST node that models any block of zero or more statements within braces. Unless they are followed by a single instruction, most control statements (if, while, for, etc) contain a block node that encapsulates their body.

In the presence of a block node, the algorithms for splitting or completing expressions are extended with a special case: it is possible to extract two or more complete subtrees, as long as they are consecutive children of the same block node. Indeed, these trees would just correspond to consecutive statements.

4.1.4 Summary

To summarize, the proposed clone detection algorithm performs the following steps:

• Parse the source code into an AST, and resolve the bindings by building the Class Graph;

• Create a post-order traversal view of the AST nodes into a list;

• Run the LZ77 algorithm on the created list, without actually encoding the duplications, but just recording them;
Resolve artifacts of the use of the LZ77 algorithm, namely the presence of clones spanning incomplete, or more than one expression. This is done using one of the following schemes:

- Splitting the clones into multiple parts, such that each part can be extracted;
- Completing the clones with additional nodes so that they can be extracted.

In both cases, a clone is either a complete subtree of the AST, or complete subtrees that are consecutive children of the same block node.

The clones are “stored” as AST subtrees. The next step, depending on the goal, could be:

- Run the “extract method” refactoring on the detected clones to perform clone extraction. This is discussed in Chapters 5 and 6.
- Highlight the detected clones in the source code for reporting purposes. This is easily done when origin tracking is implemented by the AST, as discussed in Section 2.2.4.

### 4.1.5 Complexity

The complexity of the proposed algorithm for clone detection is given by $O(n \log(n))$ on average when completing expressions, and $O(n)$ on average when splitting expressions, where $n$ is the size of the AST, and is therefore directly linked to the size of the source code.

- Parsing of Java source code runs in linear ($O(n)$) complexity [73].
- The post-order traversal and the creation of the corresponding list run in $O(n)$ complexity as well [17].
- The LZ77 algorithm runs in average $O(n)$ complexity. See Section 4.1.2.
- The complexity of completing expressions is $O(n \log(n))$ and that of splitting expressions is $O(n)$. They are detailed below.

When completing an expression, it is necessary to first look for the lowest common ancestor of the nodes, and then to include all missing descendants. Looking for the common ancestor $\Psi$ of a set of nodes $N = \{n_0, n_1, ..., n_{I-1}\}$ is implemented as follows:

\[
\Psi := n_0 \\
\text{for each } n_i \text{ in } N - \{n_0\} \\
\Psi := LCA_2(\Psi, n_i),
\]

Where $LCA_2$ is the lowest common ancestor of two nodes, and runs in $O(\log(n))$ [17].

We first note that the total number of nodes in all detected clones is limited by the size $n$ of the AST. For each of these nodes that is part of a detected duplication, it is necessary to apply $LCA_2$ with the corresponding $\Psi$, which takes $O(\log(n))$ time. The global complexity is hence limited by $O(n \log(n))$.

Then, completing the subtree with the missing descendants is implemented by creating a new sub list of nodes using a tree traversal starting from the lowest common ancestor. While this is done for every clone, the total number of visited nodes is again limited by the size $n$ of the AST.

To implement the logic of splitting expressions efficiently, the tokens detected by the LZ77 algorithms must first be put in a hash table. Then a second tree traversal can be used to mark all nodes that have all their descendants in the hash table. The extraction of the resulting expressions then requires a tree traversal for every node that has all its descendants in the hash table; however, the number of visited tokens in all traversals of that last step is limited by the total number of tokens $n$. Hence splitting expressions can be done in $O(n)$ average complexity.
4.1.6 Comparison with other approaches

The clone detection algorithm described in the previous sections shares some similarities with other existing clone detection techniques, but also has several differences. We review the main existing approaches and compare them with the presented one.

String Based Approaches

There are some similarities with string based approaches. In particular, looking for duplications in string based approaches is typically done with algorithms that are close to the LZ77 algorithm presented in Section 4.1.2, and usually involves hash tables as well.

The main difference of our approach is that it distinguishes between variables with the same name but different scopes, and hence achieves 100% precision. Like string-based approaches, white space, comments and superfluous constructs such as additional parentheses or package prefixes are ignored, but this is done without introducing artifacts.

Regarding speed, both approaches have an $O(n)$ complexity. However string based approaches are expected to be faster by a constant factor, as they do not need the parsing and traversal steps.

Token Based Approaches

Token based approaches are the closest to our approach. Replacing source code elements by abstract tokens is basically what is done as the first step of a parsing process: the lexical analysis. If we only consider the list view of the AST constructed by our algorithm (see Section 4.1.1), this list view is basically a tokenized version of the source code and is indeed close to what a token-based approach would produce.

There are notable differences though:

- Token based approaches directly build a token list from the source code, and hence do not use an AST or a Class Graph.
- Token based approaches may also produce artifacts such as those discussed in Section 4.1.3. However, only our algorithm is capable of resolving them, because it makes use of the full AST structure for that purpose.
- Token based approaches usually use the same abstract tokens for several language constructs. As a consequence, they may produce false positives. On the other hand, this abstraction allows the detection of clones that only differ by elements that can be passed as parameters (such as constant expressions or variables of the same type).
- Token based approaches are expected to be faster, because they do not build the AST and Class Graph. However, they cannot cope properly with notions such as scope and binding.

AST Based Approaches

Although clone detection approaches based on the AST have been previously proposed, they are quite different from the proposed one. In our approach, duplications are detected on a token list corresponding to a post-order traversal of the AST, and the AST is only used to recover from artifacts. The comparison is also done exactly.

AST approaches proposed by Ira D. Baxter [5] attempt to directly compare subtrees of the AST against each other. To reduce the complexity, the subtrees are not compared directly, but only their hash codes. As a consequence, this approach behaves differently than ours in the following respects:
• Complexity is higher despite of the optimization through the use of hash codes ($O(n^2)$ instead of $O(n^3)$ [5]);

• Comparison of the subtrees using hash code can produce false positives. Hence, potential candidates have to be checked again afterwards to filter out false positives.

• The use of hash code allows for some tricks, namely the detection of duplicated code in which the order of the operands of a commutative operator are swapped.

Graph Based Approaches
Graph based approaches (using the CFG, PDG or DFG) do not only abstract the syntax, but also much of the semantics. They allow the detection of various non-trivial clones, in which for instance

• The order of independent statements is not the same;

• Variables with different names that have the same semantics;

• Expressions are promoted.

These detections are possible thanks to the expressiveness of graph representations of the code such as the CFG or the PDG, and to the possibility of performing formal transformations and analyses on these graphs. However, the cost of building these graphs makes the process harder to implement, and slower (in term of algorithmic complexity [51]) than our approach.

4.2 Adaptation to Differentiation
Section 4.1 presented a new clone detection technique. In this section, we show how the technique can be adapted to the problem of differentiating the bodies of two methods. The purpose is to use it to implement the first step of the refactoring discussed in Chapter 3, forming a template method.

4.2.1 LZ77 vs Longest Common Subsequence (LCS)
Comparing the bodies of two methods looks like searching for clones. However, the problem has several differences that prevent the direct use of clone detection techniques (although a heuristic based on clone detection is discussed afterwards in Section 4.2.4):

• Differences and duplications must only be detected between the given two bodies, and not among the whole code or within a single body.

• Clone detection is usually used as a first step to extract the code duplications. When forming a template method we are interested in extracting the code differences.

• When forming a template method, the duplications and differences are only relevant when they occur in the same order in both methods. If duplications do not occur in the same order, they can not all be left in the template method (see Section 3.2.1 for a description of the problem).

A way of coping with these issues is to replace the LZ77 algorithm by the LCS (Longest Common Subsequence) algorithm. Indeed, the LCS algorithm has the following desirable properties:

• It identifies duplications between two lists, which is exactly what is needed when forming a template method. The two lists are the token lists corresponding to post-order traversals of the ASTs of the methods bodies. Note that the algorithm can be extended to the case of $n$ lists.
• It only identifies duplications occurring in the same order. Hence there is no need to cope with the problem of ordering mentioned in Section 3.2.1.

The other parts of the algorithm basically remain unmodified. Like with LZ77, the LCS algorithm is applied to two token lists. These two token lists correspond to the post-order traversals of the ASTs of the two method bodies. As with the LZ77 algorithm, the LCS algorithm is applied to token lists that do not carry the full structural information. Therefore it will produce similar artifacts. When using the LCS algorithm, there are subtle changes in the resolution of the artifacts. They are discussed below in Section 4.2.3.

4.2.2 LCS Implementations

The LCS problem can be implemented using dynamic programming [17], and achieves \(O(kn^k)\) complexity, where \(n\) is the (average) length of the sequences, and \(k\) is the number of sequences. The case we are interested in, with two sequences, is therefore solved with \(O(n^2)\) complexity.

This is bigger than the \(O(n)\) average complexity achieved by the LZ77 algorithm, although this might still be acceptable in practice. Nevertheless this remains a drawback compared to LZ77, and we therefore present a faster heuristic in Section 4.2.4.

A lot of research and literature exists on fast implementations of the LCS algorithm. The details of these algorithms are beyond the scope of this thesis. It is worth mentioning that algorithms exist, that solve the LCS problem with a complexity lower than \(O(n^2)\). In some cases the solution is exact whereas in other cases it is suboptimal, but can still be of sufficient relevance for the task of forming a template method. Among the existing algorithms, the version of Hirschberg achieves \(O(rn + n \log(n))\), where \(r\) is the length of the actual longest common subsequence. The version of Hunt and Szymanski achieves \(O(M \log(n))\) where \(M\) is the number of matches (where a match is given by a sequence of consecutive matched tokens). For a detailed survey refer to the work of L. Bergroth, H. Hakonen and T. Raita [7].

4.2.3 Application to AST Differentiation

As for the case of clone detection, the application of the LCS algorithm to two token lists produces artifacts in the result. The reason is the same as with the LZ77 algorithm, namely the fact that the token list, unlike the AST, does not exhibit the full code structure. To illustrate the problem, let us review the following code that was presented as an illustration of the “form template method” refactoring in Section 3.1:

```java
// In class Rotator
public void rotateAt(Point center, double amount) {
    translate(mult(center, -1 + 5));
    rotate(amount, center);
    translate(center);
    normalize(amount);
}

// In class Skewer
public void skewAt(Point center, double amount) {
    translate(mult(center, -1 - 4));
    amount = skew(amount);
    translate(center);
    normalize(amount);
}
```
As specified by the refactoring’s preconditions, we assume that the two methods belong to two different classes that both extend the same abstract class. As discussed in Section 3.1, this is not a hard precondition, but this is the most convenient situation for applying the refactoring.

Figure 4.3 shows the ASTs corresponding to the two methods. Performing a post-order traversal of each AST yields the two following token lists:

- \([\text{center, } -1, 5, +, \text{mult, translate, amount, center, rotate, center, translate, amount, normalize}]\)

- \([\text{center, } -1, 4, -, \text{mult, translate, amount, amount, skew, =, center, translate, amount, normalize}]\)

Applying the LCS algorithm to these two lists reveals that the following non-trivial\(^2\) sublists are duplicated:

- \([\text{center, } -1]\)
- \([\text{mult, translate, amount}]\)
- \([\text{center, translate, amount, normalize}]\)

These sublists correspond to the tokens that are the same in the two ASTs. Recall that to form a template method, it is necessary to extract all the code fragments that are different. Hence, we are not directly interested in the detected duplicated tokens, but in the remaining tokens. They actually consist of the following sublists (first line is the sublist for the first AST, and second line is the corresponding sublist for the second AST):

\(^2\)Duplicated sublists of a single token are filtered out.
As for clone detection, the sublists do not correspond to expressions that can be safely extracted. The problem can be solved in the same ways, by either splitting or completing the expressions, as discussed in Sections 4.1.3 and 4.1.3. There is a major difference though.

In case of clone detection, splitting expression was a better alternative than completing expression. Indeed, completing expressions has the drawback of adding code elements that differ in the clone, making its extraction harder and potentially impossible. Splitting expressions on the other hand may remove duplicated elements from the clones, but do not prevent their extraction (unless a clone becomes too short to be worth extracting).

When forming a template method, what is important is to leave only duplicated elements in the template method, and to extract all differences.

As we are interested in extracting code elements that differ, completing expressions may eventually include duplicated statements in the extracted method. However, unlike with clone detection, this is not a problem when forming a template method. The sole hard constraint is that only duplicated elements remain in the template method. On the other hand, splitting expressions can add non-duplicated elements in the template method by removing them from the fragments to extract. This is hence not a good solution.

Completing expressions in the detected sublists corresponding to differences yields the following new sublists:

\[-1, 5, +\], \[center, rotate\]
\[-1, 4, -\], \[amount, skew, =\]

These sublists all correspond to expressions that can be safely extracted. Indeed, applying the method extractions then produces the following template method:

```java
public void templateMethod(Point center, double amount) {
    translate(mult(center, d1()));
    amount = d2(amount, center);
    translate(center);
    normalize(amount);
}
```

Splitting expressions on the other hand would produce the following new sublists:

\[5\], \[center\]
\[4\], \[amount, skew\]

These sublists are not only too short to be worth extracting, but extracting them is not sufficient to be able to form the template method. Some differences (such as the “+” and “-” operators) remain in the method bodies.

These examples show that, when forming a template method, the technique of completing expressions is more attractive than the technique of splitting expressions.

### 4.2.4 Heuristics

Until now, we have used the LCS (Longest Common Subsequence) algorithm to find differences and duplications between the two method bodies. While this algorithm is optimal in that the returned sequence is the longest, it is not always optimal in term of refactoring. It also has a higher complexity than the LZ77 algorithm. Consider for instance the following two artificial token lists:
The LCS algorithm would return “[a, c, e, g, i]” as the longest common subsequence. In terms of refactoring though, this sequence is not ideal because almost none of its tokens are consecutive. Extracting the unmatched tokens would involve the extraction of many small methods. Furthermore, the sublist “[X1, X2, X3, X4]” has not been “detected”. Indeed, its length is actually smaller than that of the longest common subsequence. As a result, it will be extracted as a difference when forming a template method (also see Section 3.2.1 about the problem of matched statements in different order).

The problem is that LCS finds the longest common subsequence regardless of whether the resulting tokens are consecutive or not. When forming a template method, one may also want to optimize the lengths of the sublists consisting of consecutive matched tokens. This would potentially increase the length of the methods to extract and reduce their number.

Therefore, the following heuristic can be used in place of the LCS algorithm:

- Run the LZ77 algorithm on one token list, using the other one as a sliding window. This finds all duplications consisting of consecutive tokens.
- Gather the duplications starting from the longest one, and filtering out those that are reversed with at least one previously gathered duplication.

With the previous example, the duplications “[X1, X2, X3, X4]” and “[g, i]” are found (eventually, individual tokens might be matched as well if duplications of a single token are considered). Then the longest duplication, “[X1, X2, X3, X4]” is gathered first. When the second duplication, “[g, i]” is considered, it appears that it is reversed with the previous duplication: it occurs after it in the first list, but before it in the second list. Hence, it is filtered out and only “[X1, X2, X3, X4]” is finally kept.

In practice, there is no clear advantage of using either the LCS or the LZ77 algorithm (with the additional aforementioned step). LZ77 clearly favors large duplications (long sublists of consecutive tokens), but this is sometimes at the detriment of multiple smaller duplications that may eventually cover more code, despite of the largest number of methods to extract. Note that if all duplications occur in the same order, the two algorithms usually give comparable results. Differences between the two algorithms mainly show up when duplications occur in different orders.

In both cases, the resolution of artifacts further affects the results of both algorithms. A possible strategy is to apply both algorithms, resolve the artifacts for each of them, and then choose the one that permits the largest number of code duplications to be removed. An even better strategy is to use LZ77 to find all duplications (including those that are in reversed orders), and then to display them to the user so she can choose the most relevant ones; as duplications are picked by the user, reversed duplications can be automatically filtered out.

### 4.3 Renamed Variable Matching

Until now, we have shown how statements can be compared in a way that abstracts from irrelevant syntactical elements.

In this section, we go one small step toward the recognition of semantically equivalent statements, namely the recognition of variables that have the same semantics, but different names.

#### 4.3.1 The Problem with Clone Detection and Removal

The presence of variables with similar semantics but different names is not really a problem in the context of clone detection. Once a potential clone has been detected, the two occurrences are then compared again
using an exact comparison in order to filter out false positives. At that time, it suffices to match variables according to their order of occurrence. Indeed, assuming that the two code fragments are equivalent, variables that occur at the same place must be equivalent. If the match is not possible, there are two possibilities:

- The clone is simply rejected as a false positive;
- The variable occurrences that cannot be matched are passed as additional arguments. If the clone performs write accesses to them, they must be passed by reference. This is the solution taken by R. Komondoor [51]. However, he notes that this situation seldom occurs in practice. He also notes that the matching constructed by the occurrence order is not always optimal, especially if the clone is inexact and still contains several differences.

Note that the matching of variables only applies to the local variables, that is, variables that are either declared within the method body, or that are formal arguments of the method. This does not apply to fields, that are uniquely determined by their name and scope.

4.3.2 The Problem when Forming a Template Method

The problem of matching renamed variables is harder when forming a template method because:

- As for clone detection, it is much easier to match variables when we know what parts of the code are duplicated and what parts are not.
- Finding duplications however, requires us to know which variables are matched. The reason is that unlike for clone detection, the same matching of variables must hold for all the matched fragments, that is, for the entire template method.

The problem is best illustrated on the following example:

```java
public int method1(int x, int y, int z) {
    x += y + z;
    print(x);
    store(x);
    y = x - 4;
    print(y);
    return y;
}

public int method2(int x, int y, int z) {
    y += x + z
    print(y);
    store(y);
    y++;
    print(y);
    return y;
}
```

A clone detection tool that handles renamed variables would find two clones: one comprising the three first lines of the bodies, and one comprising the last two lines of the bodies. In the first clone, the variable \(y\) of the first method is matched to the variable \(x\) of the second method and vice versa. In the second clone though, the variable \(y\) of the first method is matched to the variable \(y\) of the second method.
This does not prevent the two clones from being extracted as long as the correct variables are passed as arguments.

In the case of forming a template method though, we do not extract duplicated code but different code. In particular, the fourth line of the two bodies must be extracted. Unfortunately, the remaining lines cannot be left safely in the template method because the roles of \(x\) and \(y\) are not constant during the whole body.

There are several ways of working around this problem:

- Add arguments to differentiate between the different variable matches:

  ```
  public void templateMethod(ref int a, ref int b, ref int c, int z) {
    a += b * z;
    print(a);
    store(a);
    b = d1(a, b);
    print(c);
    return c;
  }
  ```

  Here, \(a\) and \(b\) correspond to the original \(x\) and \(y\) which are matched with each other, while \(c\) corresponds to the original \(y\) that is matched with itself. Note however that the variable \(c\) in this example corresponds to either \(b\) for the first original method, and to \(a\) for the second original method. To ensure proper aliasing, it is hence necessary to pass all arguments by reference. The result of this aliasing is quite dirty programming style.

- Add an artificial delegate method whose purpose is to swap the two variables in one of the two methods. This is however artificial and hence undesirable because it adds a new method that is unrelated to the original code.

- Keep in the template method only the longest duplicated part. In this example, only the three first lines would be kept. In the general case though, it is possible that more than one duplicated part can be kept, as long as all of them have the same variable matching.

The third solution, although it leaves some duplication, is the most appealing as it does not introduce tricks that might be viewed as bad programming style, or simply as modifications that go beyond the sole application of the refactoring. However, it requires the algorithm to find the “best” matching between the variables. The “best” matching is the one that allows the highest amount of code to be kept in the template method. As the best matching may cover more than one duplication, we cannot simply take the matching of the largest duplication (although this might be considered as a fast and fair heuristic). We now introduce a new possible solution to the problem.

### 4.3.3 A Solution Using Bipartite Graph Matching

The proposed solution is to proceed in three steps:

- Compare the two method bodies without regard to local variable names. Only their data types are compared;

- Use this first comparison to find out the best match between the variables;

- Compare the two method bodies again, this time using the match computed in the previous step.
In the first step, more code than what can be left in the template method might be found: as discussed in the previous sections, some duplicated portions may not exhibit a valid variable match, or two duplications may exhibit different matches between the variables.

Finding the Best Match

The second step of the algorithm iterates on the token lists corresponding to the matched portions of the bodies. With the previous example, these token lists are the following (the first line corresponds to the first method and the second line to the second method):

\[
[x, y, z, *, +=, x, \text{print}, x, \text{store}], [y, \text{print}, y, \text{return}]
\]
\[
[y, x, z, *, +=, y, \text{print}, y, \text{store}], [y, \text{print}, y, \text{return}]
\]

Observe that (by construction) the token lists corresponding to the first and second methods only differ by the names of the local variables.

Using these token lists, a weighted bipartite graph is built:

- Nodes of the left side of the graph correspond to variables of the first method;
- Nodes of the right side of the graph correspond to variables of the second method;
- Edges are present whenever two variables are matched at least once;
- The weight of an edge is given by the number of times the two corresponding variables are matched.

Figure 4.4 illustrates the weighted bipartite graph corresponding to the given example.

Once the bipartite graph is built, the best matching between variables can be chosen by applying the maximum weighted bipartite matching algorithm on it [31]. This will keep only at most one edge per node in such a way the sum of the edge’s weights is maximized. The result of applying a maximum weighted bipartite matching on the graph of Figure 4.4 is illustrated in Figure 4.5.

The result is the expected one: namely \( x \) is matched with \( y \) and vice versa, despite two matches of the variable \( y \) with itself.

In the third step finally, the two method bodies are compared again with this matching. As a result, only the three first lines of code will be matched, and left in the template method.
There are a few subtle points to watch for when creating the template method:

- Before any differences are extracted, it is eventually necessary to rename the variables of one of the two methods so that they match the corresponding variables of the other method. In the previous example, it is necessary to rename \( x \) as \( y \) and vice versa in one of the two methods. This is basically about applying the “rename variable” refactoring multiple times.
- The arguments must eventually be reordered so that corresponding variables occur in the same order in the two methods. This is an application of the “change method signature” refactoring. Note that this step involves code modifications at every place where the method is invoked.

It is also possible to do these two steps just after the optimal matching has been computed. Then, the second time the two bodies are compared, the local variables can be matched directly by their name rather than using the computed matching.

**Correctness**

The presented algorithm implicitly makes the following assumption: the more frequently two variables are matched, the more probably they are equivalent. If we take this assumption for granted, the algorithm indeed produces an optimal result.

There are many cases however, in which this assumption might not be correct:

- An expression such as \( x \times x \) contains two occurrences of the variable \( x \), whereas the expression \( \text{square}(x) \) contains only one occurrence of it. The first expression may contribute twice to the same matching whereas the second one contributes to it only once. However, the two expressions are equivalent (assuming the \text{square} method indeed returns the argument multiplied by itself).
- While it might happen for instance that the best match associates a variable \( t \) with a variable \text{firstItem} and a variable \text{temp} with a variable \text{firstElement}; the names of the variables clearly suggest another mapping to be more relevant.
- Variables that occur more often have more weight. However the number of occurrences has no direct relationship with the “importance” of a variable in practice. In particular, temporary variables (for example counters such as “\( i \)”) usually have a weak importance, yet they are expected to occur frequently in the code.
As only the programmer can tell what the best match between variables is, there is no automatic and
optimal solution to this problem. This is a feature that is shared by most code analysis tools and smell
detectors\(^\text{3}\): they produce correct results according to some underlying assumptions (defined by metrics),
but the result may slightly (and greatly in rare occasions) differ from the user’s appreciation [2, 21].

For these reasons it is frequently considered as important to leave room to the user for changing what
is detected by the algorithm. When matching variables for instance, the result should ideally be proposed
to the user, who can either confirm it, or propose an alternative.

In practice though, complex variable matches rarely occur. Indeed, if some code is copied and pasted,
and several variable renaming are applied on the copy, there is still a unique variable matching. The
presence of other potential matchings can only occur within portions of code that appear to be duplicated,
but are only duplicated by “accident”. This concerns for instance small code fragments corresponding
to frequent constructs or design patterns. While some code might appear to be duplicated, it occurs in
different contexts and is usually not worth considering as real duplication.

Complexity

Good implementations of the maximum weighted bipartite matching algorithm run in \(O(n^2 \log n + ne)\)
where \(n\) is the number of nodes and \(e\) is the number of edges [31]. In our problem, \(n\) is hence the number
of variables and \(e\) is the number of different matches between variable pairs. It is not clear whether
the number of variables \(n\) grows linearly with the size of a method, however this can be expected from
the number of distinct matches \(e\) when arbitrary methods are considered. In all case, \(n\) and \(e\) are both
bounded by the size of the methods.

A possible heuristic that achieves \(O(e \log e)\) complexity is to use a greedy approach. The idea is just
to iteratively pick edges in order of decreasing weight from the bipartite graph. Each time an edge is
picked, other edges incident on either of the two nodes are removed.

With the previous example, the edge from \(x\) to \(y\) is picked first, and the edge from \(y\) to \(y\) is removed. Then the two remaining edges (from \(y\) to \(x\) and from \(z\) to \(z\)) are picked. The result of this heuristic on this
example is the same as the weighted bipartite matching. In practice, the heuristic seldom differ from the
exact algorithm, which is also due to the fact that most of the time the best matching is mostly constant
within the whole method body.

The heuristic either picks or removes an edge in each iteration, and is hence bounded by the number
of edges. The \(O(e \log e)\) cost is hence only due to the necessity of sorting the edges by their weights.

4.4 Summary

This chapter was devoted to the problem of comparing executable source code such as method bodies.
We proposed a novel approach that provides enough abstraction in order to be insensitive to irrelevant
syntactical sugar, while still providing high performances and 100% precision. Both an algorithm for
clone detection and an adaptation for code differentiation were proposed, and were compared against
existing approaches. The proposed algorithm was then further improved in order to cope with renamed
variables.

\(^3\)This name covers all kind of tools that attempts to automatically detect badly written code, and eventually to suggest a refac-
toring for improving the code.
Chapter 5

New Approaches to Flow Analysis

Chapter 4 has dealt with the problem of comparing statements, both for clone detection and to form a template method. This chapter deals with another analysis that is required for the next step of the process of forming a template method, namely the extraction of the methods.

In general, it is not always possible to extract an arbitrary list of consecutive statements into a new method. There are various preconditions. If one of them is not verified, the method extraction cannot take place. The preconditions for a method extraction heavily depend on the underlying programming language. For the Java programming language for instance, a code fragment can only be extracted under the following preconditions [19, 77]:

- The fragment is properly enclosed in a block: it does not cross either the beginning or the end of a block such as the beginning or end of a loop or conditional without crossing both;
- The fragment only includes full statements and expressions. No statement or expression is only partially included. This precondition and the above one are already verified by the statement comparison technique discussed in Chapter 4.
- Execution can only leave the fragment in a single place; this is not verified for instance if the fragment contains a `break` statement that does not exit to the end of the fragment;
- The execution can only enter the fragment at a single place. This precondition and the previous one are also known as the `single entry, single exit` precondition;
- The fragment does not make use of a nested class declared outside of it, and does not declare a nested class used outside of it;
- The resulting method has to return at most one value\(^1\).

Note that other languages may not require some of the preconditions, or may require other ones. Languages that allow arguments to be passed by reference for instance allow the extraction of methods that need to return more than one value. Some visual languages such as ProGraph [61] and FOOD [44] permit the extraction of methods where the execution can leave the method in multiple possible places.

With the C language, the use of macros allows the extraction of almost any sequence of code, including partial expressions, because macros are not interpreted by the compiler, but just replaced by their actual value before the compilation starts.

\(^1\)This holds if we want to perform a straightforward extraction. Section 6.3 discusses how to work around this limitation using more complex extraction processes.
We assume at this stage that we exactly know what code fragments we want to extract. What is described is a new method to analyze the data and control flows, and to check for the various preconditions of method extraction. The purpose of the data flow analysis is to identify the variables that need to be passed as arguments to the extracted method, and those that need to be returned as results. The purpose of the control flow analysis is to check whether or not the method to extract has a single entry and a single exit execution path. Multiple exit execution paths can show up for instance if the statements to extract contain break or continue statements.

The remainder of this chapter is structured as follows: Section 5.1 illustrates the problem on concrete examples. Section 5.2 defines the notions used in the proposed analysis algorithm. Section 5.3 shows how the result of the analysis can be used to identify the arguments and results of the method to extract. Section 5.4 describes in details how the analysis is actually implemented. Section 5.5 shows how the analysis algorithm can be used to check preconditions. Section 5.6 finally compares the proposed algorithm with other related approaches.

5.1 Motivations

When extracting a method, it is necessary to figure out four things:

- The incoming data flows, that is, the variables that need to be passed as arguments to the extracted method.
- The outgoing data flows, that is, the values that need to be returned as results by the extracted method.
- The outgoing control flows, that is, the different locations in the code where the execution of the code fragment to extract can end. While the execution typically ends after the last statement to extract, it can also end on a return or break statement.
- The incoming control flows, that is, the different locations in the code where the execution of the code fragment to extract can start. There can be more than one incoming control flow if we want to extract statements from before a loop to somewhere in the middle of it.

It is clear that depending on the results of these analyses, it may happen that the method cannot be extracted (or at least not easily). This is the case for instance if:

- There is more than one incoming control flow.
- There is more than one outgoing control flow. Note that the extraction can still be possible by applying a few transformations of the code; this is addressed by Section 6.2.
- There is more than one outgoing data flows. This is only a limitation with languages such as Java, in which a method cannot return more than one value and does not support passing arguments by reference. As with multiple outgoing control flows, applying a few code transformations may still permit the extraction; this is discussed in Section 6.3.

Data and control flow analyses are usually done by first building CFG, DFG or PDG of the code. Using such graphs, gathering incoming and outgoing control and data flows consists of straightforward graph operations, which makes this approach appealing. Another advantage of graph-based approaches is that they provide much more information than what is actually needed for method extraction. As such, they can also be used for various purposes, such as for compiler optimizations [32].

Unfortunately, the construction of the CFG, DFG or PDG is a complex and tedious task, with a super linear complexity (some worst cases for control flow analysis are exponential [25, 51, 76]). This
is expected to be too heavy for the sole purpose of refactoring, which only requires a small part of the information provided by these graphs.

Indeed, inspection of the source code of the Eclipse Java development environment by the authors, namely the implementation of the “extract method” refactoring, revealed that it does not make use of such a structure to analyze data and control flows. Instead, an ad-hoc algorithm is used. Unfortunately, the used algorithm is flawed, as explained in the next sections.

We now introduce a new algorithm for collecting incoming data and control flows directly from the AST, and therefore without requiring the construction of the CFG, DFG or PDG. A particularity of the proposed algorithm is that it only provides the information that is required for method extraction, and hence runs much faster.

5.1.1 Examples

Before describing the algorithm, let us discuss the problem of figuring out the arguments and results of a method to extract.

It seems at a first glance that it suffices to identify all variables that are read to get the arguments, and all variables that are written to get the results. While this undoubtedly produces a correct result, the result is suboptimal: it may find more arguments and more results than actually required. It is intuitively possible to do better:

- A variable that is read may not need to be passed as an argument if it is first written by the extracted method. In that case, the last value of the variable before entering the method is never actually used.
- A variable that is written by the method does not need to be returned as a result if it is never read afterwards by the calling method, or if the calling method gives it a new value before it ever reads it.

Furthermore, keeping the number of arguments and results to the minimum is desirable to keep the extracted method’s signature as simple as possible. In particular, for languages such as Java that do not allow a method to return more than one result, it is crucial to keep the number of results to a minimum.

Just looking for read and write accesses is simple to implement, but this may hence produce a suboptimal result. This may also prevent easy method extraction in the presence of more than one result, although some results might not actually be necessary.

However, getting an optimal result, that is, identifying only the arguments and results that are actually necessary is a more challenging problem. Let us illustrate the problem with a concrete example:

```java
void difficult(int x, int y) {
    tt(y);
    while (x < 0) {
        tt(x--); // extract from here...
        y++; // ... to here
        if (y >= 0) {
            x = y - 1;
        } else {
            break;
        }
    }
    doStuff(y);
}
```

Assume we want to extract the two lines with comments. What variables do we need to pass as argument, and what variables do we need to pass as result?
A naive search for read and write access would figure out that both variables (x and y) are both read and written\(^2\). Hence it would suggest that they both need to be passed as argument and returned as result. However, a closer look reveals that the new value of x is never read again after the extracted method, meaning at a first glance that only y needs to be actually returned.

Now consider a small modification of the code, where the else branch of the conditional has been removed:

```java
void difficult(int x, int y) {
    tt(y);
    while (x < 0) {
        tt(x--); // extract from here...
        y++; // ... to here
        if (y >= 0) {
            x = y - 1;
        }
        doStuff(y);
    }
}
```

It may seem that nothing has changed: both variables are read and need to be passed as arguments, and only the new value of y seems to be read after the extracted method. Regrettably, this is wrong: in the modified version of the code, both variables need to be returned! The reason is that the new value of x is potentially read in the next iteration of the while loop. More precisely, its value is read in the “\((x < 0)\)” control expression, and also by the “\(x--\)” statement.

An interesting observation from this fact is that we also need to consider variable accesses occurring before the code fragment to extract to figure out the required results, when in presence of loops.

While x does not need to be returned in the first version of the code, the reason is not just that the x variable is not read after the statements to extract. In fact it is necessary to analyze the two possible execution paths of the conditional to figure out whether or not the modified value of x is actually needed:

- Either the if condition is verified, and x is assigned a new value. In that case the value modified by the two statements to extract is never read, even if the loop iterates again.
- Or the if condition is not verified, and the break statement exits the while loop. In that case, the loop never iterates again and the modified value of x is never read.

In fact, if we remove the while loop from the second version of the code (where both x and y are required results), the situation changes again and only y needs to be returned.

These simple examples show that the required arguments and results are not only determined by the read and write accesses, but also by the control flow of the statements. The notion of control flow denotes the topology of the execution paths, and is determined by loops (while, for, etc), conditionals (if, switch) and jumps (break, return, exceptions, etc). See Section 2.1.3 for a discussion of the control flow.

### 5.1.2 Requirements

From the above example, we need the following information to solve the problem:

- All variable accesses, including those that are before or after the code fragment to extract.
- All control statements such as loops, conditionals and jumps.

\(^2\)Note that the “++” and “--” operators require the variable to be first read, and then written.
The Program Dependence Graph (PDG) or the (CFG, DFG) pair give us all this information directly. If one of them is available, analyzing the incoming and outgoing control and data flows is a trivial task.

However, the construction of any of these graphs is an expensive task, much more than the construction of the AST. As a refactoring tool is used during the development cycle of an application, it is expected to respond quickly.

Also observe that the graph models provide much more information than what we actually need. Indeed they provide the full control and data flows at any part of the code. When extracting a method though, we are only interested in the data and control flows at two specific locations of the code: just before (incoming flows) and just after (outgoing flows) the code fragment to extract.

The next section presents a new method to analyze the control and data flows. Unlike the PDG or the (CFG, DFG) pair, it computes only what is required, namely the incoming and outgoing flows. Furthermore, it only makes use of the AST and the Class Graph, and runs in linear time.

5.2 Definitions

The proposed flow analysis algorithm makes use of regions and flags. Regions are consecutive portions of the code, whereas the flags are Boolean attributes. The flags are defined for every combination of a region and a local variable.

Note that in the remainder of this section, we assume that the bindings are available for all variables, which means that variables with the same name but different scopes are properly differentiated, using the Class Graph for instance.

Three regions denoted by the \( b, f \) and \( a \) symbols, and four flags denoted by the \( R, W, w \) and \( L \) symbols are used. We first define these flags and regions in the next section. Then Section 5.3 shows how these flags can be used to identify arguments and results. Finally Section 5.4 presents an algorithm for collecting the value of these flags from the code.

5.2.1 Regions

The regions correspond to three portions of the entire body of the method from which we want to extract a code fragment. They are defined as follows:

- “\( b \)” is the portion of code before the code fragment we want to extract;
- “\( f \)” is the code fragment we want to extract;
- “\( a \)” is the portion of code after the code fragment we want to extract.

These regions are defined based solely on the statements order in the code, and ignore the dynamic execution order.

5.2.2 Flags

The flags are Boolean values defined for each combination of a variable and a region. They are defined as follows:

- \( R \) means that the variable is read, or conditionally read in the corresponding region. A variable is “conditionally” read if the read access occurs for instance in only one branch of an if-else construct, or within a loop that may not iterate at all.
- \( W \) means that the variable is certainly written in the corresponding region.
• $w$ means that the variable is conditionally written in the corresponding region.

• $L$ means that the variable is read or conditionally read before it is certainly written in the corresponding region. In other words, it means that the value of the variable before entering the region is eventually used.

For each variable, 12 Boolean values are hence collected, corresponding to all the combinations of flags and regions. A Boolean value for a given variable will be noted using the region in subscript. For example, $R_a$ denotes a read access of the variable before the code fragment to extract, while $w_a$ denotes a conditional write access of the variable after the code fragment to extract.

There is no flag to denote a variable that is declared but never used (neither read nor written). Such a variable just has none of the flags that is set to true, as if it did not exist at all.

Note that we consider that the $W_b$ flag is true for all variables corresponding to arguments of the method regardless of the result of the analysis, even if there is no explicit write access before the fragment to extract. The reason is that arguments are always given actual values before the method’s body is executed. The values are the actual parameters supplied by the calling method.

The meaning of the $W$ flag is subtle, because it depends on the location within the code we are considering. The exact definition of this flag is that, assuming the execution has reached a given location in the code, the variable is certainly written up to that point. The locations of interest in our case are the ends of the different regions $a$, $f$ and $b$. If a write access only occurs in an if statement (without “else”) for instance, this write access is considered as conditional after the if statement. However, within the if block, it is considered as certain. Indeed:

• If we assume the execution has reached the “then” block of the if statement, the write access is unavoidable. In this case the $W$ flag is set.

• If we only assume the execution has reached some place after the if statement, we cannot tell whether the “then” block was executed or not. In this case only the $w$ flag is set.

When the $L$ flag is true for a given variable and region, we say that the variable is live in that region. This notion of “liveness” has a link with the DFG: when a variable is live in a region, an edge corresponding to that variable enters the subgraph corresponding to that region in the DFG.

Intuitively, if a variable is live at a certain point in the code, it means that its current value will potentially be used at a later point in the code. If it is not live, it means that its current value will no longer be used (either because the variable is never read again, or because it is assigned a new value before it is read again).

In compilers, the notion of live variables plays an important role. If a variable is live at a given point in the code, it means that a register (or a memory location) must be allocated to maintain its value. Correctly identifying variables that are no longer live allows the compiler to reuse the corresponding registers (or memory locations) for other variables, and hence to optimize the registers’ usage.

### 5.3 Using the Flags

For a given variable, simple Boolean expressions based on the flags can be used to determine whether the variable is a required argument or result. However, the flags can also be used for other purposes. We first illustrate how the flags can be used to identify the required arguments and results.

#### 5.3.1 Arguments

To check whether a given variable is a required argument, it suffices to check whether the $L_f$ flag is true for that variable. Indeed, the variable must be passed as argument if its value before the fragment is used
in the fragment. Its value is used not only if it is read in the fragment, but the read access must occur before a write access gives it a new value. This is exactly what is captured by the $L_f$ flag.

5.3.2 Results

To check whether a given variable is a required result, checking the $L_a$ does not suffice: although this flag means that the value of the variable is used after the fragment to extract, there is no need to return it as a result if the fragment to extract does not modify its value at all. The following expression hence has to be used:

$$L_a \land (W_f \lor w_f) \quad (5.1)$$

Basically, the additional condition says that the variable is actually modified (either conditionally or certainly) within the fragment to extract. If only $L_a$ is true, but not $(W_f \lor w_f)$, it means that the variable was only given a value before the fragment to extract.

There is an interesting observation to make at this point. To determine the required arguments, we only used the $L_f$ flag. We did not put as an additional condition the fact that the variable must actually be modified before the fragment to extract. In other words, we just used the $L_f$ flag instead of the following expression, which is symmetrical to the expression (5.1):

$$L_f \land (W_b \lor w_b) \quad (5.2)$$

The reason the $(W_b \lor w_b)$ part is not necessary is that it must always hold. Indeed, if the variable is not given a value before the fragment to extract and the $L_f$ flags is true, it would mean that the variable is read before it is initialized. In Java, this is actually a compile error. However, for other programming languages that give default values to uninitialized variables, then the expression (5.2) would be more appropriate than the $L_f$ flag alone.

5.3.3 Declaration Absorption

In some cases, a variable is no longer used in the original method after the code fragment has been extracted. In that case, the variable declaration can safely be moved from the original to the extracted method. This situation can be identified by the following expression:

$$\neg (R_b \lor W_b \lor w_b) \land \neg (R_a \lor W_a \lor w_a) \quad (5.3)$$

This expression is basically the negation of all possible accesses before and after the fragment to extract. Note that if the variable is an argument of the original method, the $W_b$ flag is true by definition, and this expression does not hold. This is relevant because removing an argument is not directly possible: it involves adapting all the callers and is in fact another refactoring. This is not part of the method extraction process.

5.3.4 Declaration Duplication

In other cases, a variable is used by both the original method and the extracted method, but is not passed as argument. This occurs for instance if the extracted method assigns a new value to the variable before it ever reads it. In that case, the declaration of the variable must be duplicated in the extracted method. The variable is then declared in both the extracted and the calling method.

This particular situation can also be identified by a Boolean expression using the flags:

$$\neg L_f \land (W_f \lor w_f \lor R_f)$$

$$\land (W_b \lor w_b \lor R_b \lor W_a \lor w_a \lor R_a)$$

88
The first line states that the variable is not a required argument \((\sim L_f)\) but is used (read or written) in the fragment to extract \((W_f \lor w_f \lor R_f)\). The second line states that the variable is used as well (read or written) outside of the fragment to extract. Note that the expression can be simplified by removing some of the \(R\) flags, because a read access cannot be present unless the variable is first initialized \((W\) flag).

Observe that the flags can also be used to reveal some “spurious” programming practices. For instance, if \((W_b \lor w_b) \land \sim (R_f \lor R_a)\) is true, it means that the variable is written but never read afterwards! When extracting a method though, it is not our task to “cleanup” such code (although it might be relevant to issue a warning to the user).

### 5.3.5 Other Results

At a given point in the code, a local variable may have been declared, but not initialized. By “initialized”, we just mean that a value is given at least once to the variable for sure (not only conditionally). In Java, an uninitialized variable cannot be passed as an argument or result of a method (a compile error is issued in such a case). However, Java allows the presence of uninitialized variables as long as they are never read. We now prove that, with our flag based analysis and expressions, an uninitialized variable is never identified as a required argument or result by our algorithm, under the assumption that the code has no compile errors.

We start with the identification of arguments. Let us assume that some variable is detected as a required argument and is not initialized before the fragment to extract. A variable is identified as an argument if \(L_f\) holds. By definition of the \(L_f\) flag, this implies that \(R_f\) is true, and that the variable is not written for sure in the fragment to extract before the first read access in that fragment. As we assume no compile error, then \(W_b\) must be true, or else the variable cannot be read. However, if \(W_b\) is true, it means that the variable is initialized before \(L_f\) is set, which is a contradiction of our initial assumption.

This proves that our algorithm will never identify an uninitialized variable as a required argument. Hence we do not need to deal with uninitialized variables explicitly.

The same derivation can be used to show that an uninitialized variable is never identified as a required result. A variable is identified as a result if \(L_a \land (W_f \lor w_f)\) holds, which implies that \(L_a\) holds. Again by definition of the \(L_a\) flag, this implies that \(R_a\) is true and that the variable is not written for sure after the fragment before the first read access after the fragment. Assuming no compile error, it follows that one of \(W_b\) or \(W_f\) must be true, or else the variable cannot be read. However, if \(W_b \lor W_f\) is true, it means that the variable is initialized before \(L_a\) is set, which is again a contradiction to our initial assumption.

### 5.4 Gathering the Flags

We now describe how the values of these flags can be gathered from the code for all variables. Section 5.4.1 first presents the basic process, in the absence of any control statements (loop, conditional, jumps). Section 5.4.2 then refines the process in order to deal with conditionals. Section 5.4.3 further refines the process in order to deal with loops. Finally Section 5.4.4 completes the process in order to deal with jump instructions. The whole algorithm is summarized in Section 5.4.5.

As previously mentioned, the algorithm is based on the AST representation of the code.

#### 5.4.1 The Basic Process

In the case the method body contains no control statements such as conditional and loops, the analysis is quite simple. It suffices to perform a post-order traversal of the AST. As local variables are accessed, the \(R\) and \(W\) flags are set as appropriate for each of the three regions. The \(L\) flag can be set on the fly as well: on a given read access to a variable, it has to be set if and only if \(W\) is not yet set for that variable.
Using a post-order traversal ensures that all expressions are visited in their execution order, with only a few exceptions (such as assignments, where the right hand side must be handled before the left hand-side). These exceptional cases can easily be handled appropriately with minor modifications in the implementation.

5.4.2 Dealing with Flow Breaks

This simple algorithm needs to be extended in order to properly deal with complex control flows. Here, we propose a first extension of the algorithm. The purpose is to make it handle all blocks where the execution is conditional. This also includes loops that may not iterate at all. We do not yet deal with loop iterations at this stage. This is covered later on in Section 5.4.3.

The Process

The proposed solution is to identify split points, merge points, and the corresponding execution paths.

Split points occur whenever the execution flow can diverge, and create more than one possible execution path. An if-then-else construct for instance is a split point, and creates two execution paths: one for the “then” block and one for the “else” block. If there is no “else” block, we still have two execution paths, but one of them is empty (it covers no statements, and corresponds to the case in which the code execution skips the conditional). The beginning of a while loop is also a split point: one path traverses the loop body and the other one skips it and is empty. The latter corresponds to the case in which the loop never executes. Note that a do loop on the other hand always iterates at least once, and hence is not a split point. Indeed, the body of a do loop is guaranteed to execute at least once.

Note that we consider the body of a loop, or the “then” and “else” parts of a conditional as blocks. In the source code, a block is usually formed by statements placed inside braces (“{” and “}”). Loops and conditionals in Java allow a single statement without braces instead of multiple statements within braces. However we also consider such a single statement as a block.

Merge points are the locations in the source code where multiple execution paths converge. In a conditional for instance, the paths for the “then” and the “else” blocks are typically merged just after the conditional. But this is not always the case: if one of the two blocks contains a break statement, then the merge point of the corresponding path is after the loop that is escaped.

The handling of conditionals works as follows: When a split point is encountered, the current flags table is stored. Then, each path is analyzed separately with a new copy of the current flags table. When the merge point is reached, the table stored at the split point is merged with every table whose merge point is located at this place. The merging logic is defined below.

We use the “k” superscript to identify the flags of the tables to merge (corresponding to the different execution paths) and the lack of index to identify the flags of the table stored at the split point. The latter is the one that is modified by the merging logic, and then used to continue the analysis with the statements following the merge point. The merging logic is expressed as follows:

\[
\begin{align*}
L &= L \lor (\neg W \land \bigcup \langle L^k \rangle) \\
R &= R \lor \bigcup \langle R^k \rangle \\
W &= W \land \bigcap \langle W^k \rangle \\
w &= w \lor \bigcup (W^k \lor w^k)
\end{align*}
\]

(5.4) (5.5) (5.6) (5.7)

Let us explain these equations:

- A variable gets live if it was already live (\(L\)), or if it was not previously written (\(\neg W\)) and is live in at least one of the paths (\(\bigcup \langle L^k \rangle\)).

3Assuming no break statements, which are discussed later in Section 5.4.4.
• A variable gets read if it was already read, or if it is read in at least one of the paths. Note that $R$ includes conditional read accesses.

• A variable gets certainly written if it already was, or if it is certainly written in all paths.

• A variable gets conditionally written if it already was, or if it is written (conditionally or certainly) in at least one of the paths.

The $L$ flag requires the most complex update and deserves some more explanations. First it is clear that if the variable was live before the split point, it remains live regardless of the different paths. In case it was not live when reaching the split point, there are two possibilities:

• Either the variable is already written for sure when reaching the split point, in which case it cannot possibly get live, regardless of the different paths. It can only possibly get live if it has not yet been written for sure, hence the "¬$W$" expression.

• Or the variable has not yet been written for sure when reaching the split point. In that case it gets live if it is live in at least one of the different paths ($\bigcup (L^k)$).

We also note that the $w$ flag might be set even if the write access is certain (for instance if it is certain in all paths). In practice this is not a problem because the $W$ flag always "overrides" the $w$ flags when it is true.

**Examples**

Before we discuss how to deal with loops, let us illustrate the algorithm on the following example:

```c
int test(int y) {
    int x;
    if (y > 0) { // extract from here...
        x = 1;
        y = y + x;
    } // ... to here
    return y;
}
```

This code has one split point and one merge point. The split point is at the beginning of the conditional’s block whereas the merge point is at the end of the conditional. The split point creates two different paths: one traverses the block’s statements, whilst the other skips them. The former corresponds to the case in which the condition evaluates to true and the latter corresponds to the case in which it evaluates to false. Note that the "$(y > 0)$" expression is always evaluated and is hence located before the split point. The split point is actually located on the opening brace.

When reaching the split point, only the $W_b$, $R_f$ and $L_f$ flags are true for the $y$ variable. At this point, the current flags table is stored, and a copy of it is created to analyze the content of the block.

Just before the end of the block, $R_f$ and $W_f$ are true for both the $x$ and $y$ variables. The $L_f$ is true for the $y$ variable (recall that the read access on the right hand side of the assignment is processed before the write access on the left hand-side). However, $L_f$ is not true for the $x$ variable because a certain write access occurs before the first read access. It is important to note that we are processing the conditional’s block only at this stage, and the write access is certain from within that block.

In parallel, another copy of the flags table is done at the beginning of the block. This copy corresponds to the case in which the conditional’s block is not evaluated. Hence this copy is not modified at all and only has the $W_b$, $R_f$ and $L_f$ flags of the $y$ variable set to true, and none for $x$. 

91
When the merge point is reached, the stored table is combined with the two tables corresponding to the two paths according to equations 5.4. Replacing the $k$ index with the two paths, these equations become:
\[
\begin{align*}
L &= L \lor (\neg W \land (L^1 \lor L^2)) \\
R &= R \lor (R^1 \lor R^2) \\
W &= W \lor (W^1 \land W^2) \\
w &= w \lor (W^1 \lor w^1 \lor W^2 \lor w^2)
\end{align*}
\]
Replacing the flags with their actual values for the $x$ variable and the region within the fragment to extract, we get:
\[
\begin{align*}
L &= \text{false} \lor (\neg \text{false} \land (\text{false} \lor \text{false})) = \text{false} \\
R &= \text{false} \lor (\text{true} \lor \text{false}) = \text{true} \\
W &= \text{false} \lor (\text{true} \land \text{false}) = \text{false} \\
w &= \text{false} \lor (\text{true} \lor \text{false} \lor \text{false} \lor \text{false}) = \text{true}
\end{align*}
\]
In other words, just after the conditional
\begin{itemize}
  \item $x$ is not live;
  \item $x$ is possibly read;
  \item $x$ is not certainly written. This means for instance that it is considered as uninitialized by the compiler at this point, despite the fact it was considered as initialized when the “$y = y + x$” statement was encountered; the reason is that the “$x = 1$” statement is within the conditional.
  \item $x$ is conditionally written.
\end{itemize}
Replacing the new values of the flags in equation 5.2 reveals that only $y$ is a required argument. Tests with the NetBeans 6.5 Java development environment revealed that it is fooled by this example, and incorrectly identifies $x$ as a required argument. Here’s the caller method after the method extraction performed by NetBeans:
\begin{verbatim}
int test(int y) {
  int x;
  y = extracted(x, y);
  return y;
}
\end{verbatim}
The code does not compile, because $x$ is passed as an argument although it is uninitialized. In the original method, the first write access to $x$ is certain from the point of view of the inner block, but not from the point of view of the outer block, which may fool NetBeans.

Now consider the following modified version of the code (two statements have been added after the block to extract):
\begin{verbatim}
int test(int y) {
  int x;
  if (y > 0) { // extract from here...
    x = 1;
    y = y + x;
  } // ... to here
\end{verbatim}
x = y;
y = y + x;
\textbf{return} y;
}

The beginning of the flags analysis is the same as with the first version. Continuing the flags analysis on this new version until the end of the method’s body, observe that the $R_a$ and $W_a$ flags are true for $x$, but not $L_a$. For $y$, $R_a$, $L_a$ and $W_a$ are all true.

Replacing the values of these flags in equation 5.1 reveals that only $y$ is a required result, but not $x$. Tests with Visual Studio 2005 (using the C# language) revealed that it incorrectly identifies both $x$ and $y$ as required results. Here’s the method extracted by Visual Studio:

```c
int extracted(ref int y) {
    int x;
    if (y > 0) {
        x = 1;
        y = y + x;
    }
    \textbf{return} x;
}
```

The code has a compile error on the \textbf{return} statement, because the $x$ variable is not initialized. Note that Visual Studio returns $x$ as the method’s result, and $y$ using and argument passed by reference. NetBeans on the other hand now refuses to extract a method, complaining that two variables need to be returned.

Here’s the correct result as produced by our implementation:

```c
int test(int y) {
    int x;
    y = extracted(y);
    x = y;
    y = y + x;
    \textbf{return} y;
}
```

```c
int extracted(int y) {
    int x;
    if (y > 0) {
        x = 1;
        y = y + x;
    }
    \textbf{return} y;
}
```

The resulting code compiles without any errors and is a correct method extraction.

5.4.3 Dealing with Loops

We now introduce improvements of the algorithm so that it also properly copes with loop iterations.

Motivation

Until now, we have introduced three regions: the region before the fragment to extract (identified by the $b$ letter), the fragment to extract itself (identified by the $f$ letter), and the region after the fragment to extract
(identified by the a letter). Until now, these three regions were only determined by the location of the statements. The b region comes first, followed by the f region, followed by the a region.

However, in the presence of loops, these notions are not sufficient. Consider the following example:

```c
int firstFibbGreaterOrEqualTo(int v) {
    int x = 0;
    int y = 1;
    while (x < v) {
        int t = y;
        y = y + x; // extract this line
        x = t;
    }
    return x;
}
```

Assume that we want to extract the line with the comment “extract this line”. There are two problems with the definitions of the regions as proposed until now:

- Although the “int t = y;” statement occurs before the fragment to extract, it is also potentially executed after it, in the next iteration in case the loop iterates more than once. The same holds for the “(x < v)” condition of the while loop.
- Although the “x = t;” statement occurs after the fragment to extract, it is also potentially executed before it, in the previous iteration if the loop iterates more than once.

The “Entrance”

Loops are handled in two steps:

- A new region, the entrance, is defined; and flags are gathered separately on this region;
- The flags gathered in the entrance are combined with those of the a region (the region after the fragment to extract). The definition of this latter region is hence implicitly extended.

The entrance region is defined as follows:

- It starts at the beginning of the top-most loop that surrounds the fragment to extract, and includes any control expressions that are evaluated on every iteration.
- It ends at the end of the fragment to extract.

Here’s the above code, with comments saying what line belongs to what regions (the entrance is referred to with the e letter):

```c
int firstFibbGreaterOrEqualTo(int v) {
    int x = 0; // [b]
    int y = 1; // [b]
    while (x < v) { // [b, e]
        int t = y; // [b, e]
        y = y + x; // [f, e]
        x = t; // [a]
    } // [a]
    return x; // [a]
}
```

4Observe that for the first time in this thesis, we present source code that really does something meaningful :)

94
Intuitively, the *entrance* corresponds to all the statements that are potentially affected by the fragment to extract in the next iteration, in case the loop iterates more than once. Accesses to variables in the *entrance* must hence be considered in a way similar to accesses in the region *after* the fragment, but only in a conditional way.

Note that there is at most one entrance because only the top-most loop enclosing the fragment to extract needs to be considered. Loops that do not include the fragment to extract do not require any special handling. Loops that are not top-most are implicitly included by the analysis of the top-most loop. When the beginning of the entrance is encountered, two copies of the current flags table are done. One is left as is, and the other one is used to analyze the entire entrance region.

When the analysis of the loop is terminated, flags of the *a* region are combined with the two copies using the merging logic presented by equation 5.4.

The idea underlying this process is to consider the entrance as an additional conditional block that occurs just after the loop. Here’s an illustration of the idea in form of pseudo code (although this is not what is actually done: the code itself is not modified during the analysis):

```c
int firstFibbGreaterOrEqualTo(int v) {
    int x = 0;
    int y = 1;
    while (x < v) {
        int t = y;
        y = y + x;
        x = t;
    }
    if (iterates_more_than_once) {
        // These are the statements of the entrance:
        (x < v);
        int t = y;
        y = y + x;
    }
    return x;
}
```

Without introducing the entrance region, we note that the flag $R_a$, $W_a$, $w_a$ and $L_a$ are all false for the $y$ variable. Indeed this variable is neither read nor written after the line to extract. With the proposed loop handling, two new tables of flags are created: one corresponds to an empty block and also has these flags set to false. The other one corresponds to an analysis of the entrance, and has the flags $R_e$, $W_e$ and $L_e$ set to true for the $y$ variable (The $e$ index denotes the entrance region). These two tables of flags are used to update the one of the region after the fragment according to equation 5.4.

As a result of this update, the flags $R_a$, $w_a$ and $L_a$ get true. A consequence is, for instance, that $y$ gets a required result of the method to extract. Indeed, its new value is required for the statements “int $t = y$” and “$y = y + x$” in case the loop iterates more than once.

If the *while* loop were replaced by an *if* statement, this would not be the case.

The “Exit”

One may ask oneself if it is also necessary to introduce a notion complementary to the *entrance*, or an *exit* region. This region would start from the beginning of the fragment to extract, and stop at the end of the top-most loop. Like for the *entrance*, it would be handled like a conditional statement occurring before the loop. Here’s an illustration in pseudo-code with our previous example:

```c
int firstFibbGreaterOrEqualTo(int v) {
```
In theory this could be done. In practice however, this is useless, at least when the Java language is considered. The reason is that the arguments are determined only by the $L_f$ flag. Hence they do not depend on the read and write accesses that occur before the fragment to extract!

The exit region is only necessary if the underlying programming language allows the use of uninitialized variables (and gives them default values). In that case, as stated in Section 5.3.2, equation (5.2) should be used instead. This modified equation also requires the $W_b$ and $w_b$ flags, corresponding to accesses before the fragment to extract. For such programming languages, it is necessary to consider the exit region as well and to use it to update the $b$ region. The handling of the exit region is similar to that of the entry region and is therefore not detailed.

**Examples**

To illustrate the algorithm again in presence of loops, consider the following example:

```java
void test(int x, int y) {
    while (x < v) {
        int t = y;
        y = y + x;
        x = t;
    }
    return x;
}
```

In theory this could be done. In practice however, this is useless, at least when the Java language is considered. The reason is that the arguments are determined only by the $L_f$ flag. Hence they do not depend on the read and write accesses that occur before the fragment to extract!

The exit region is only necessary if the underlying programming language allows the use of uninitialized variables (and gives them default values). In that case, as stated in Section 5.3.2, equation (5.2) should be used instead. This modified equation also requires the $W_b$ and $w_b$ flags, corresponding to accesses before the fragment to extract. For such programming languages, it is necessary to consider the exit region as well and to use it to update the $b$ region. The handling of the exit region is similar to that of the entry region and is therefore not detailed.

**Examples**

To illustrate the algorithm again in presence of loops, consider the following example:

```java
void test(int x, int y) {
    while (x < 0) {
        doStuff(x--); // extract from here...
        y++; // ... to here
    }
    System.out.println(y);
}
```

Consider the `x` variable. If we ignore the loop, only the flags $R_b$, $R_f$, $W_f$, $L_f$ are true. Hence the variable `x` is not identified as a required result.

If we analyze the *entrance*, which is formed by the following expressions and statements:

```java
{x < 0}
doStuff(x--);
y++;
```

The flags $R_e$, $W_e$ and $L_e$ are set to true. Combining these flags with those of the region after the fragment to extract, we observe that $R_a$, $w_a$ and $L_a$ gets true. Using this new result, the `x` variable is now correctly identified as a required result.

The reader can verify that both `x` and `y` are required arguments and required results.

Interesting is the following modified version of the code:

```java
void test(int x, int y) {
    while (x < 0) {
```
In this modified version, the $W_a$ flag is true for the $x$ variable before the entrance is handled. When the flags of the entrance are combined with those of the region after the fragment to extract, only $R_a$ and $w_a$ gets true. $L_a$ remains false, precisely because $W_a$ is true (see equation 5.4). As a consequence, the $x$ variable is no longer identified as a required result in this modified version of the code.

This is indeed the correct result: in this modified version, there is no need for the extracted method to return the $x$ variable, because it is given a new value just after the extracted statements. Tests revealed that the Eclipse 3.4 and NetBeans 6.5 development environments are both fooled by this situation: they both complain that method extraction cannot be done because they erroneously detect both $x$ and $y$ as required results, although only $y$ is actually required. Visual Studio makes the same mistake, however it still permits method extraction (in C#), by passing the variables by reference.

5.4.4 Jump Statements and Exceptions

In this section, we discuss how the flags analysis handles jump statements such as break, continue, return and throw.

Forward Jumps

We first discuss the break statement, which is an instruction that moves the execution forward in the code. First consider the following code, with a loop and a conditional:

```java
while (a != b) {
    // path A
    if (a > b) {
        a = a % b; // path C
    } else {
        b = b % a; // path D
    } // merge point for C and D
} // merge point for A and B
```

The different execution paths and their merge points have been highlighted in comments. The while loop generates two execution paths. The first one (path A) traverses the loop body whereas the second one (path B) skips it. Both paths are merged at the end of the loop.

The if statement generates two paths as well. The first one corresponds to the “then” block (path C) and the second one to the “else” block (path D). Both paths are merged at the end of the conditional.

A break statement is handled as follows: when a break statement is encountered, the merge point of the current path is moved from its current location to after the end of the escaped loop.

Consider the following modified version of the code as an illustration:

```java
while (a != b) {
    // path A
    if (a > b) {
        a = a % b; // path C
    } else {
        b = b % a; // path D
    }
}
```

97
break; // merge point of D moved after end of loop
} // merge point for C only
} // merge point for A, B and D

The result of the break statement is that now only path C (the “then” branch of the conditional) has its merge point at the end of the conditional. The merge point of the “else” branch (path D) has been moved to the end of the loop. As a consequence, the end of the loop is now the merge point of three different paths, and the end of the conditional is the merge point of a single path.

A detailed example involving a break statement is given later in Section 5.4.6.

Backward and Outside Jumps

The continue statement is similar to the break statement, but moves the merge point of the current path back to the beginning of the loop. However, it cannot be handled in the same way: As the AST is traversed from the beginning to the end, this merge point will never get a chance to be handled, because the corresponding node has already been traversed when the continue statement is encountered.

To handle it, the merge point is not moved to the beginning of the loop, but to its end. Unlike with the break statement, the merge point is not placed just after the loop, but inside it, after its last body statement. Furthermore, that location is also marked as the merge point of the current path at that location. The following example illustrates the process:

```java
do {
1     // path A
2     if (a > b) {
3         a = a % b; // path C
4     } else {
5         b = b % a; // path D
6         continue; // merge point of D moved before end of loop
7     } // merge point for C only
8     a = a - 1;
9     // merge point for current path and path D
} while (a != b); // merge point for A and B
```

Observe that although a do loop always executes at least once, the a variable is not written for sure after the loop. The case in which it is not written occurs when all loop iterations take the path D and reach the continue statement. In this particular case, lines 3 and 8 are never executed.

The case of the return statement is simpler to handle: when a return statement is encountered, the merge point for the current path is just dropped. Indeed, a return statement jumps outside of the method’s body, and hence the corresponding merge point would never be reached by an analysis of the method’s body.

This can be somehow confusing at a first glance, as shown by the following example:

```java
void test(int x, int y) {
1     while (x < 100) { // extract from here
2         x = x * 2;
3         y = y - 1;
4     } // to here
5     if (y > 0) {
6         return; // path A
7     } else {
8         x = y; // path B
9     }
```
In the absence of the return statement, Line 9 would be the merge point of paths A and B. Because of the return statement, the merge point of the path A is dropped, and only the merge point of the path B remains. A consequence is that the $W_a$ flag is set for the $x$ variable (and not just the $w_a$ flag) when the merge point is reached, because the assignment of Line 8, although in a conditional, is considered as certain. Indeed, path B is the only path that has its merge point at Line 9.

This can be confusing because it seems at a first glance that $x$ is not necessary written, precisely because the return statement may terminate the execution of the body before the assignment.

To understand it, recall that the values of the flags correspond to a given location in the code that the execution is assumed to have reached. The fact the $W_a$ flag is true does not mean that “$x$ is certainly written” at that location, but means that “assuming that the execution has reached that location, $x$ is certainly written”.

The result of the flag analysis is indeed correct: because the $W_a$ flag is set for $x$, the $L_a$ flag never gets set, and $x$ is hence not a required result of the method to extract.

The correct code after the method extraction looks as follows:

```java
void test(int x, int y) {
    y = extracted(x, y);
    if (y > 0) {
        return; // path A
    } else {
        x = y; // path B
    } // merge point of B only
    doStuff(x);
}

int extracted(int x, int y) {
    while (x < 100) {
        x = x * 2;
        y = y - 1;
    }
    return y;
}
```

Observe that the value of $x$ modified by the extracted method is not returned, and is indeed never reused by the calling method. Eclipse 3.4 and NetBeans 6.5 are both fooled by this example. They identify both $x$ and $y$ as required results and complain that they cannot extract a method that has to return two values.

**Exceptions**

Exceptions are hard to handle, and even in other areas (such as detecting uninitialized variables), the underlying logic is often approximated in a “safe” way. “Safe” means that the worst case is always considered, and hence errors are reported rather than producing wrong results in cases where the logic cannot be inferred exactly.

To understand the problem, consider the following code:

```java
void test(int x, int y) {
    try {
```
if (y == 0)  
    throw new Exception();  
  x = process(a[y]);  
  x = 10;  
}  
catch (Exception ex) {  
ex.printStackTrace();  }

// is x written for sure at this location?

Consider the question of whether the above code performs a certain write access to the x variable, or only a conditional write access.

The presence of the throw instruction at line 3 makes the answer obvious: the write access is only conditional.

However, even if Lines 2 and 3 were removed, the write access would still be conditional, but this is now less obvious. The reason is that line 4 may also generate an exception before the x variable is ever given a value. An exception can be thrown at line 4 for several different reasons:

- the process method itself throws an exception;
- the value of the y variable is out of the range of the a[] array;
- a is null.

We first consider how to handle the throw statement. Basically, the statement is just handled either as a break or as a return statement, depending on whether the thrown exception is caught by the method or not. If the exception is caught, the throw statement is like a jump to the beginning of the catch block. Hence the proper handling is to move the merge point of the current path to the beginning of the catch block. If the exception is not caught, the throw instruction acts as a return statement: the execution of the method terminates. In that case, the merge point of the current path is just dropped, like for a return statement.

The throw instruction is the only one that always throws an exception. However, almost all other statements can potentially throw an exception, as discussed on the previous example. Instructions that can potentially throw an exception can be treated as follows:

- If the exception is caught by the method, two paths are created (as in a conditional) corresponding to whether or not the exception is actually thrown. One path corresponds to the execution continuing normally, and the other to the execution jumping to the beginning of the catch block. Both paths have their merge point after the end of the catch block.
- If the exception is not caught by the method, no action is taken. This case is similar to a conditional return statement.

While the handling of exceptions is not very different than the handling of conditionals, break and return statements, the actual implementation has to face with several challenges:

- To know whether an exception is caught or not, it does not suffice to have access to the catch instruction, because a given exception class is also indirectly caught if its parent class is caught. It is hence necessary to have access to the full class hierarchy of the different exception classes.
- To know whether a given method called by the body being analyzed can throw a given exception, it is necessary to have access to the signature of that method, and hence to the Class Graph.
• To know whether an instruction can throw a runtime exception (such as null references), it is necessary to know, for each individual instruction of the language, the list of runtime exceptions that can eventually be thrown.

• To know whether a given method called by the body being analyzed can throw a given runtime exception, it is necessary to analyze the body of that method and the bodies of all methods it directly or indirectly invokes.

Because of the last two points, runtime exceptions are usually too hard to handle correctly in practice. While the other exception types could be handled without too many difficulties, it is a common practice to just approximate the behavior of exceptions in a safe way. A “safe” approximation is just to consider that the entire body of a try block may or may not execute. In other words, a try block is just handled like an if block, and the catch blocks are treated like else if and else branches.

Most existing Java compilers actually use this approximation when checking whether a variable is initialized or not. The same approximation was hence used in our implementation so that it remains consistent with the Java compiler. However, explicit throw statements are handled exactly as discussed previously.

5.4.5 Summary

To summarize, the algorithm used to identify the arguments and results of a method to extract works in two steps. First, tables of Boolean flags are gathered for each local variable by traversing the AST. Then the arguments and results are identified using Boolean expressions on the flags.

During the first step, a table of Boolean flags is maintained for each local variable, and each of the three regions: before the fragment to extract, within it and after it. There is one full AST traversal to do and optionally one additional partial traversal for the entrance region, in case the fragment to extract is within a loop. Only the top-most loop containing the fragment to extract has to be considered; hence there is at most one entrance to analyze.

During a traversal, the Boolean flags are updated as variable accesses are encountered. In addition, the following actions are taken when appropriate:

• When a split point is encountered (begin of conditional, loop or try block):
  – The different execution paths are determined;
  – The merge points for the different paths are recorded in a queue for future handling;
  – The current table of flags is stored with the merge point;
  – New copies of the current tables of flags are created and used to analyze each of the different paths.

• When a break statement is encountered:
  – The merge point for the current execution path is moved from its current location to the target of the break statement, that is, just after the end of the escaped loop.

• When a continue statement is encountered:
  – The merge point for the current execution path is moved from its current location to just before the end of the corresponding loop.

• When a return statement is encountered, the merge point of the current execution path is dropped.

• When a throw statement is encountered:
– The merge point of the current execution path is dropped if the thrown exception type is not caught by the method;
– Else, the merge point of the current execution path is moved to the begin of the corresponding catch block.

• When a merge point is encountered (end of conditional or loop, etc):
  – The tables of flags stored with the merge point is merged with the table of flags that were used to analyze the different execution paths.

• When a loop is encountered, that encloses the fragment of code to extract:
  – A region of the code, the entrance, is identified. It covers the code from the beginning of the outermost loop to the end of the fragment to extract;
  – A new copy of the current table of flags is created and used to analyze the entrance region;
  – A second copy of the current table of flags is created, and is left unmodified;
  – The entrance is analyzed and updates one of the two copies;
  – When the end of the loop is reached, the two copies are merged with the current one as for a conditional with no “else” branch.

Once the traversal is finished, the arguments and results are identified using Boolean expressions involving the values of the flags gathered by the previous step.

5.4.6 A Complete Example

Let us illustrate the entire algorithm line by line on the following example:

```c
void difficult(int x, int y) {
  tt(y);
  while (x < 0) {
    tt(x--); // extract from here...
    y++;     // ... to here
    if (y >= 0) {
      x = y - 1;
    } else {
      break;
    }
  doStuff();
}
```

Before the first line is encountered, the flag \( W_b \) is already set for both the \( x \) and \( y \) variables, as these are arguments.

**Line 1.** In the first line, a read access to the \( y \) variable is performed. Hence the \( R_b \) flag is set for that variable.

**Line 2.** Here, a loop is encountered. However, the loop’s control expression, “\((x < 0)\)” is considered first, because it is also executed first, and unconditionally. It actually sets the \( R_b \) flag for the \( x \) variable.

Then the actual loop is considered. As it may not iterate at all, it generates two different execution paths: one traverses the loop body, while the other skips it. At this point, the current table of flags
is stored, and two copies are created (for each execution path). The tables of flags have the following values: \{R_b, W_b\} for \textit{x} and \{R_b, W_b\} for \textit{y}. The corresponding merge point for both tables of flags is the end of the loop, at line 11.

In the remaining statements of the loop body, only one of the two copies will be updated, as the other one corresponds to the execution skipping the loop. The copy that will be updated becomes the current (or active) table of flags.

At the same time, we have to consider and analyze the \textit{entrance} region corresponding to the loop. The analysis of the \textit{entrance} is not very different than the analysis of the rest of the code, and is therefore not detailed. The result of this analysis will be considered when discussing line 11.

**Line 3.** Here, we are entering the region within the fragment to extract. The \texttt{--} operator sets both the \(R_f\) and \(W_f\) flags for the \textit{x} variable. As the \(R_f\) flag is set first (the variable is first read), the \(L_f\) flag is set as well because the \(W_f\) flag is not yet set at this time. The current table of flags hence has the following value for the \textit{x} variable: \{\(R_b\), \(W_b\), \(R_f\), \(W_f\), \(L_f\)\}.

**Line 4.** Here, the \(R_f\), \(W_f\) and \(L_f\) flags are both set for the \textit{y} variable.

**Line 5.** This is a conditional. However, the conditional expression, \((y \geq 0)\) is considered first because it is also executed first, and unconditionally. As we are now in the region after the fragment to extract, the \(R_a\) and \(L_a\) flags are set for the \textit{y} variable.

Then the conditional itself is considered. This is a split point generating two different execution paths, for the “then” and “else” blocks. The current table of flags is stored, and two new copies are created. The first copy will be used in the “then” branch, and the other one in the “else” branch. The merge point for the tables is the end of the conditional, at line 9.

Note that at that time, two tables of flags are stored. The first one was stored when the beginning of the \texttt{while} loop was encountered, and the other has just been stored when the \texttt{if} conditional was encountered.

**Line 6.** This line sets the \(R_a\) flag for the \textit{y} variable. This has actually no effect as it is already set in the current table of flags. The line also sets the \(W_a\) flag for the \textit{x} variable.

**Line 7.** Here, the current table of flags is switched to the second copy that was created when the \texttt{if} instruction was encountered, as we are entering the “else” branch.

**Line 8.** Wow, this is a \texttt{break} instruction. Its consequence is to change the merge point of the table of flags corresponding to the “else” branch. The merge point becomes the end of the escaped loop, namely line 11. As a result, only the table of flags corresponding to the “then” branch will be merged with the stored one at the end of the conditional, at line 9. On the other hand, three different tables will be merged at the end of the loop (line 11): the one corresponding to the loop body, the one corresponding to the execution skipping the loop, and finally the one corresponding to the “else” branch, because of the \texttt{break} statement.

**Line 9.** Here, the table of flags of the “then” branch is merged with the one that was stored at the beginning of the conditional. As previously stated, the table of flags corresponding to the “else” branch is not included. The table of flags for the “then” branch has the following values: \{\(R_b\), \(W_b\), \(R_f\), \(W_f\), \(L_f\), \(W_a\)\} for \textit{x} and \{\(R_b\), \(W_b\), \(R_f\), \(W_f\), \(L_f\), \(R_a\), \(L_a\)\} for \textit{y}. The table of flags stored when the conditional was encountered has the following values: \{\(R_b\), \(W_b\), \(R_f\), \(W_f\), \(L_f\)\} for \textit{x} and \{\(R_b\), \(W_b\), \(R_f\), \(W_f\), \(L_f\), \(R_a\), \(L_a\)\} for \textit{y}.

After applying the merging logic discussed in equation (5.4), the resulting table of flags has the following values: \{\(R_b\), \(W_b\), \(R_f\), \(W_f\), \(L_f\), \(W_a\)\} for \textit{x} and \{\(R_b\), \(W_b\), \(R_f\), \(W_f\), \(L_f\), \(R_a\), \(L_a\)\} for \textit{y}. This resulting table becomes the current one. There was nothing dramatic at this point, only the \(W_a\) flag has been set for the \textit{x} variable.

Note that if there were no \texttt{break} statement, the table of flags of the “else” branch would be included in the merging logic, and would result in the \(W_a\) flags to be set for \textit{x} instead of \(W_a\). Let us explain this difference:

- If there was no \texttt{break} statement, the \textit{x} variable would only be conditionally written after the conditional. Indeed, it is only accessed if the “then” branch is executed.
Finally, the table of flags that was stored at the beginning of the loop has the following values: 
\{R_b, W_b, R_e, W_e, \mathcal{L}_e\} for x and \{R_b, W_b, R_f, W_f, \mathcal{L}_f, R_a, \mathcal{L}_a\} for y. Note that the entrance stops at line 4, before the end of the loop. Hence all write accesses are considered as certain in this region. Flags of the entrance (with the e index) will be combined with flags of the region after the fragment (with the a index).

The current table of flags has the following values: \{R_b, W_b, R_f, W_f, \mathcal{L}_f, W_a\} for x and \{R_b, W_b, R_f, W_f, \mathcal{L}_f, R_a, \mathcal{L}_a\} for y.

The table of flags of the entrance has the following values: 
\{R_b, W_b, R_e, W_e, \mathcal{L}_e\} for x and \{R_b, W_b, R_f, W_f, \mathcal{L}_f, R_a, \mathcal{L}_a\} for y. Note that although the table of flags of the entrance has the \mathcal{L}_e flag set for x, this flag is not set in the final result because the W_a flag was already set in the current table for x. In the absence of the break statement, this would change as only the W_a would be set for x.

Observe that the handling of the entrance has set the \mathcal{L}_a flag for the x variable and the W_a flag for the y variable. The \mathcal{L}_a flag for x is set because the variable is potentially read in the next iteration. However, \mathcal{L}_a is not set because W_a was set at line 6, and preserved at line 9.

Line 11 is also the merge point for three different tables of flags:

- The table corresponding to the execution skipping the loop body. It has the following values: \{R_b, W_b\} for x and \{R_b, W_b\} for y.
- The table of flags corresponding to the execution of the entire loop body (and hence taking the "then" branch). This is the current table of flags after merging the table of the entrance. It has the following values: \{R_b, W_b, R_f, W_f, R_a, W_a\} for x and \{R_b, W_b, R_f, W_f, \mathcal{L}_f, R_a, \mathcal{L}_a, w_a\} for y.
- The table of flags corresponding to the execution of the loop body from the beginning to the break statement. This table has the following values: \{R_b, W_b, R_f, W_f, \mathcal{L}_f\} for x and \{R_b, W_b, R_f, W_f, \mathcal{L}_f, R_a, \mathcal{L}_a\} for y.

Finally, the table of flags that was stored at the beginning of the loop has the following values: \{R_b, W_b\} for x and \{R_b, W_b\} for y.

By applying the merging logic, the resulting table of flags has the following values: \{R_b, W_b, R_f, W_f, \mathcal{L}_f, w_a\} for x and \{R_b, W_b, R_f, W_f, \mathcal{L}_f, R_a\} for y. □

\footnote{We only consider the Java language. Other programming languages allow a method to access to local variables of the calling methods, and would hence require a different handling that is beyond the scope of this thesis.}
Result of the Analysis

From these final values, the equations discussed in Sections 5.3.1 and 5.3.2 can be used to identify that

- \(x\) is a required argument (its \(L_f\) flag is set);
- \(y\) is a required argument (its \(L_f\) flag is set);
- \(y\) is a required result (\(L_a\) and \(W_f\) are set);
- \(x\) is not a required result (although \(w_f\) is set, \(L_a\) is not set).

The Eclipse 3.4, NetBeans 6.5 and Visual Studio 2005 development environments are all fooled by this complex example, and all identify both \(x\) and \(y\) as required results.

The correct resulting code after the method extraction process, as done by the proposed algorithm, looks as follows:

```c
void simple(int x, int y) {
    tt(y);
    while (x < 0) {
        y = extracted(x, y);
        if (y >= 0) {
            x = y - 1;
        } else {
            break;
        }
        doStuff();
    }
}

int extracted(int x, int y) {
    tt(x--);
    y++;
    return y;
}
```

Observation

Interesting is the role of the `break` statement in the above example. If it were removed, many things would change:

- At line 9, the flags table for \(x\) would be \(\{R_b, W_b, R_f, W_f, L_f, w_a\}\). In other words, the \(w_a\) flag would be set instead of the \(W_a\). Indeed, if the execution traverses the “else” branch it now also ends at the end of the conditional, and not at the end of the loop. The \(x\) variable is hence only conditionally written at this place.
- At line 11, the flags table for \(x\) after merging the table of the `entrance` would be \(\{R_b, W_b, R_f, w_f, R_a, w_a, L_a\}\). Here the \(L_a\) flag would be set because the \(W_a\) would not have been set in the current table.
- At line 11, after merging the tables from the three paths, the final table for the \(x\) variable would be \(\{R_b, W_b, R_f, w_f, L_f, w_a, L_a\}\). Unlike the version with the `break` statement, the \(L_a\) flag would be set.
The net effect is that removing the `break` statement would make the `x` variable a required result in addition to the `y` variable.

### 5.4.7 Other Uses

Consider the following method:

```java
int test(int y) {
    int x;
    x = y + 1; // from here
    System.out.println(x); // to here
    return y;
}
```

This code does not contain any conditional, loop or jump statement and is therefore straightforward to analyze. At the end of the analysis, the table of flags has the following values: \( \{ R_f, W_f \} \) for `x` and \( \{ W_b, R_f, L_f, R_a, L_a \} \) for `y`. Note that the `W_b` flag for the `y` variable is set because `y` is an argument.

Using the equations discussed in Sections 5.3.1 and 5.3.2, we can show that the method only requires `y` as argument, and no result.

More interesting is the use of equation (5.3) discussed in Section 5.3.3 for the `x` variable. This equation actually evaluates to `true`, because `x` is only used with the fragment to extract. As a consequence, its declaration can be moved (and not copied) from the original method to the extracted method. The two methods hence look as follows after the method extraction:

```java
int test(int y) {
    // declaration of x removed
    extracted(y);
    return y;
}
```

```java
void extracted(int y) {
    int x; // declaration of x moved here
    x = y + 1;
    System.out.println(x);
}
```

While Eclipse, NetBeans and Visual Studio all add the declaration of `x` in the extracted method, they also all leave the original declaration in the calling method. This is not an error, as the resulting code still compiles. However a warning is issued by the compiler because the `x` variable is unused in the calling method.

Removing unused declarations is not strictly considered as a part of the method extraction process; however this example illustrates another potential use of the flags.

### 5.5 Control Flow Checks

We have discussed in Section 5.3 how to use the flags in order to figure out the data flow, and in Section 5.4 how to collect the flags from the code. We now explain how to extend the process of gathering the flags in order to check for preconditions related to the control flow.

Note that one precondition was that the fragment to extract is properly enclosed in a block. It should not, for instance, cross the beginning of a block and not its end, or vice versa. This precondition is trivial
to check when traversing the AST, by counting and matching the number of entered block and the number of exited blocks. We hence only discuss the single entry and single exit preconditions.

5.5.1 Single Exit

Recall that at various points in the analysis, split points are encountered, and execution paths are created with corresponding merge points. For the method extraction to be possible, it is necessary that the execution only exits the method to a single place, namely to the statement that just follows it.

For that purpose, it is necessary to associate, with each merge point, the location of the corresponding split points. Then it is necessary to perform a check when the end of the fragment to extract is reached. At this point, the algorithm has to verify that all pending merge points whose split points are within the fragment are located at the current instruction (the one just after the fragment to extract). If at least one of these merge points is not located there, the method cannot be extracted.

In addition to the above check, while the fragment to extract is traversed, any return statement, when encountered, automatically invalidates the precondition, unless the fragment to extract ends at the end of the method body.

5.5.2 Single Entry

There is no need to check for this precondition, because it can only be broken if the fragment to extract crosses the beginning of a loop, but does not include its end, or vice versa. Indeed, in such a situation, the execution can enter the middle of the fragment when the loop iterates, or when the loop is escaped by a break instruction. However, this situation is already handled when checking that the fragment to extract is properly enclosed in a block.

Another situation which might break this precondition is the presence of a goto instruction. As the Java language does not have such an instruction, we will not address this situation.

More formally, observe that the target location of any “jump” of the execution always fall into one of the following target locations:

- The begin of a block (e.g. continue statement, iteration of a loop, thrown exception that is caught);
- After the end of a block (e.g. conditionals, break statements, loop that never iterates);
- Outside of the method (return statement, thrown exception that is not caught).

If we assume that the fragment to extract has more than one entry, it follows that the target location is not the first statement of the fragment, and hence that the fragment crosses the beginning or a block but not the end, or vice versa.

Note that if both the beginning and the end of a block are part of the fragment, the jump occurs within the fragment, but does not enter it.

5.6 Linking with Other Approaches

Although the analysis technique presented in the last section works directly on the AST, it is strongly related to the Control Flow Graph (CFG). Recall that the nodes of the CFG are the individual instructions, and the edges correspond to the different execution paths. The fragment to extract corresponds to a subgraph of the CFG.

Notions that have been introduced in the analysis have direct counterpart in the structure of the CFG; namely:

6Observe that the goto instruction is not only bad design, but would make the analysis harder to implement!
• A split point corresponds to a node of the CFG that has more than one outgoing edge. It corresponds to a location in the code where the execution can take two or more different paths.

• A merge point corresponds to a node of the CFG that has more than one incoming edge. It corresponds to a location in the code where the execution can arrive from two or more different paths. In some cases though, a merge point in our analysis can correspond to a node with only one incoming edge. This occurs for instance at the end of a conditional where one of the branches contains a break statement. In theory, such a location is not really a merge point. However, treating it as such does no harm, as it results in doing nothing (as if there were no merge point).

• Apart from the "degenerate" merge points mentioned above, an execution path in our analysis corresponds to a path in the CFG that links a node with more than one outgoing edge to a node with more than one incoming edge.

• A loop in the code corresponds to a cycle in the CFG. If the fragment to extract is inside of the loop, the corresponding nodes are within the cycle in the CFG. Let denote by \( H \) (Head) the subset of nodes of the CFG corresponding to the code from the beginning of the method to the end of the fragment to extract. And denote by \( T \) (Tail) the subset of nodes of the CFG corresponding to the code from after the fragment to extract to the end of the method. The entrance corresponds to all the nodes of \( H \) that are reachable from at least one node of \( T \). Observe that the nodes of \( H \) correspond to instructions occurring before those corresponding to the nodes of \( T \). Hence, nodes of \( H \) can only be reached from nodes of \( T \) if the corresponding instructions are inside of a loop.

These similarities suggest that our approach is not very different than the construction of the CFG. However, the process just gathers the relevant information without actually building the full CFG. The advantage of this approach is that it is faster than the full construction of the CFG. The drawback is that it only captures the information that is relevant to a particular method extraction. Hence, unlike the CFG, the collected information can hardly be reused for other purposes. There are still some exceptions, as discussed in Sections 5.3.3 to 5.3.5.

There is also a link with the DFG, namely the meaning of the \( L \) flags. Recall that the edges of a DFG correspond to data, and hence indirectly to the values of local variables. If the \( L \) flag is set for a given region and variable, it means that at least one edge corresponding to that variable in the DFG falls into a node corresponding to that region.

The notion of live variables in our analysis only considers three regions. In compiler theory, the notion is more general and is defined at every instruction. This more general definition is closely related to the DFG: for each variable that is live at a given point, there is an edge in the DFG and vice versa.

### 5.7 Coping with Rich Languages

The described algorithm has to identify various items in the code, namely:

• read accesses to local variables;
• write accesses to local variables;
• split points;
• the different paths generated by a split point;
• merge points;
• loop constructs;
• jump instructions.

In practice, identifying all these items can get more cumbersome than it seems at a first glance, especially when coping with a full featured programming language such as Java, because each item can be caused by several different instructions. For instance:

• A write access does not only occur with the assignment operator ("="), but also with the derived versions (such as "+=", "-=", etc) as well as with prefix and postfix increment and decrement ("++" and "--").

• Split points, execution paths and merge points are created by different conditional constructs (if, switch and the ternary "? :" operator), by different loop constructs (for, while), by different instructions (break, continue, return, throw) and even by every statement or expression that can potentially throw an exception!

• Several loop constructs exist: for, enhanced for (since Java 1.5), while and do.

• Several jump instructions exist: break, return, continue, throw.

The difficulty lies in the fact that all these constructs slightly differ in the detailed way they should be handled. The do loop for instance always iterates at least once, unlike the while and for loops. The for loop has three control expressions to consider, against one for the while loop. Whereas the ternary "? :" operator is simple to handle, the if statement requires the handling of optional else if clauses. The switch statement requires handling of the “fall through” situations (when the code below a label does not end with a break statement, the execution continues on the next label).

There is no magical way of coping with the complexity of a real-world programming language. If a given programming language has many similar constructs with subtle differences, it is necessary to deal with all of them. However, various programming practices can be used to simplify the task as much as possible for the flags analysis: using well-designed class hierarchies, using classifiers, and using separate class hierarchies to implement strategies.

Using Class Hierarchies

Object oriented methodology would suggest using a different class for modeling every different construct of the language. All classes would extend the same abstract class. Then the required logic is modeled as abstract methods that are implemented in every subclass according to the modeled construct. The problem with this approach is that the logic that is required for loops is not necessary required for all the other language constructs. A straightforward implementation would hence result in many empty methods. Furthermore, two constructs that are similar would share a lot of common code.

A good inheritance tree might solve the problem, but it is expected that multi-inheritance is unavoidable. Indeed there are several different concerns (loop constructs, read/write accesses, conditional execution, etc), and most Java constructs are related to more than one concern.

This could still be done quite cleanly in Java using interfaces. However, if we want to reuse an existing AST structure (which is the case for this thesis, in which we reuse the AST structure provided by the Eclipse JDT), it is expected that the inheritance tree is already given and cannot be changed for new purposes.

Using Classifiers

A second solution is to use classifiers. A classifier is basically a set of functions that is able to say whether a given construct belongs to a specific group or not. One such function, “isLoop” for instance, could be used to check whether a given construct is a loop or not. Another function, “isConditional” could be
used to check for conditionals. More important for the flags analysis presented in Chapter 5 is a function such as “isSplitPoint”, that could be used to check whether the given construct is a split point for multiple branches.

With such a classifier, there is no need to create a complex inheritance tree and to rely on the instanceof operator to check the nature of a construct. There is also no proliferation of methods such as “isLoop” within the classes corresponding to AST nodes, because this method is isolated in the classifier.

On the other hand, adding a new construct would require both the creation of a new class and the modification of the relevant classifiers. Recall however that if we rely on a specific and existing AST implementation, we have no control on the AST classes. Nevertheless, we can still cope with new constructs appearing in new versions by modifying the classifiers when appropriate.

Using Separate Class Hierarchies

The use of a classifier, as its name suggests it, only allows us to differentiate between different kinds of constructs. It is still necessary at some place to implement differences in behavior. When a loop is encountered during the flags analysis for instance (see Section 5.4.3), it is necessary to gather all the control expressions of the loop that are iterating, and those that are not. The underlying logic is obviously slightly different for each kind of loop.

As we rely on an existing class hierarchy for the AST node, we can neither adapt its structure, nor add new methods to the existing classes (at least not in a straightforward way). A solution is to create a parallel class hierarchy. Each concrete class of the class hierarchy corresponds to a concrete class of the existing AST class hierarchy, and encapsulates the additional logic that is specific for a given purpose, such as the handling of loop in the flags analysis.

In practice, more than one additional behavior are required. In our flag analysis for instance, we also need some behavior to handle loops, such as determining the entrance, but we also need some additional behavior to handle flow breaks, such as determining the paths and merge points. This can be implemented using several class hierarchies.

For each required behavior, a separate class hierarchy can be built. While this generates a large number of classes, this strategy also has advantages: because every behavior uses its own hierarchy of classes, there is no longer a need to create a single big class hierarchy for everything, which would probably require multi inheritance and be overwhelmingly complex. Furthermore, a given concern is usually not relevant for all constructs of the language. Hence only classes for the relevant constructs are actually required.

The idea is illustrated by Figure 5.1: two class hierarchies are shown. One of them implements the logic required for handling loops, whereas the other one handles the logic required for handling control flows (split points, paths, etc).

As a last point, it is necessary to link the classes together. We should somehow know that the class corresponding to the while AST node for instance corresponds to the WhileLoop class of the class hierarchy related to loops. This logic can easily be incorporated into methods of the classifiers.

Other Approaches

The use of classifiers and separate class hierarchies, as presented in the previous Sections, is one way of overcoming the fact we cannot change the existing AST class hierarchy. While we have proposed to implement separate inheritance trees for every different concern, we want to point out other possible solutions.

A possibility would be to formally and completely describe the semantics of every construct of the language. Then, the individual behaviors regarding the different concerns could be deduced from these
formal descriptions\textsuperscript{7}. An advantage of such an approach is that each construct is self-contained and self-described at a single place. On the other hand, depending on the language that is used for the description, it may not be easy to infer the required properties for a specific construct and a given concern.

Furthermore, the separate class hierarchies, as proposed in the previous Section, are just a simplified and organized version of this approach: only the construct’s semantics that are actually required are described, and they are cleanly separated by concern, using different class hierarchies.

Another possibility would be to use aspect oriented programming. Aspects provide a way of compile-time or run-time introspection (depending of the implementation), and allow one to add functionalities (methods, fields, implemented interfaces) to an existing class without modifying the source code of the class: the new functionality is added to an aspect, which is then “attached” to the target class(es) by the compiler or runtime.

\textbf{5.8 Summary}

This chapter explored a new approach for the problem of analyzing the data and control flows of code statements. The proposed method is closer to the needs of a refactoring tool, compared to existing algorithms and models that are mainly targeted to compilation. As a result, the proposed method does not allow for all the optimizations of a compiler. On the other hand, it does not require the heavy construction of a CFG or PDG from the source code. By making use of tables of Boolean flags and Boolean expressions, it is hence simpler to realize. Based only on the AST and the Class Graph, the proposed approach also provides high performances, and can rely on existing parsers for most of the work.

\textsuperscript{7}This approach has been used for the CPU emulation of the STonX Atari ST emulator \cite{82}: the CPU emulation code is automatically generated by a script that uses the CPU manufacturer’s accurate documentation of the individual CPU instructions.
Illustrative examples revealed several bugs in existing development environments, which are resolved by the proposed approach. We finished by highlighting how an implementation can cope with the richness of constructs that are present in a real-world programming language such as Java.
Chapter 6

New Approaches to Precondition Resolution

In the previous chapter, we provided an algorithm to gather the arguments and results of the method to extract. At the same time, the algorithm provides us a way of checking most of the preconditions of the “extract method” refactoring.

What should be done if one of the preconditions is not verified? When the “extract method” refactoring is considered alone, the simplest answer is just to abort the process and report an error to the user. The user can then fix the problem, and try launching the refactoring again.

However, when forming a template method, the situation is different: method extraction is only a part of the process, and multiple method extractions are usually needed. The whole process is illustrated by Figure 6.1.

Furthermore, the statements to extract are not decided by the user, but are the result of the differentiation process performed between the two initial methods. There are two consequences, a bad one, and a good one: The bad consequence is that if only one method extraction cannot be performed, the whole process of forming a template method must be aborted. Indeed the last step, the “pull up” operation relies on all differences being extracted into delegate methods (see Step (C) of Figure 6.1). The good consequence is that, because the fragments to extract are not chosen by the user, it is eventually possible to modify them in order to make the extraction possible when relevant.

The idea of modifying the fragment to extract in order to allow method extraction is usually not used by refactoring tools, but is already heavily exploited by clone detection and extraction tools. Indeed, with these tools, the fragments to extract are also determined by the tool itself (when looking for clones), and not by the user (although various degrees of user interaction are commonly implemented).

In both cases, the goal is the same as ours: we want, by modifying the fragment to extract, to resolve preconditions that are not satisfied. More precisely, given a fragment to extract and a precondition that is not verified, we want to transform the fragment into a semantically equivalent one in which the broken precondition gets verified.

While the process is similar to clone extraction, we want to propose some new techniques, whose properties and aims are somehow different, namely:

- Limited semantic abstraction, and minimal code changes. While it is fine for a clone detection tool to recognize semantically equivalent code fragments that have different structures (swapped statements, while replaced by for, etc), this is less desirable for a refactoring tool. First, changing the code too much to allow a successful refactoring may surprise the programmer. Second, any

\footnote{The precondition regarding nested class declaration, which is very specific, is not addressed.}
Figure 6.1: Illustration of the process of forming a template method.
adaptation of the code to allow a refactoring is better done by the programmer himself. Hence, only limited code changes should be investigated.

- Ability to handle to “difficult” situations. It is always frustrating for the user when a refactoring cannot be done. This goal seems somehow incompatible with the previous one at a first glance, yet it addresses a different problem. Here “difficult” does not refer to the complexity of recognizing semantic equivalence, but rather to a variety of cases in which performing method extraction (a part of the “form template method” refactoring) is not trivial (more than one exit path, more than one return value, etc). The algorithm should be able to perform method extraction even in such cases.

This section reviews the different preconditions of the “extract method” refactoring. For each precondition, there exist situations in which the precondition is broken and hence where the method extraction cannot take place. We propose, for each case, solutions that may usually (but not always) solve the problem.

6.1 Block Boundary Crossing

Method extraction cannot be performed on a fragment that crosses the beginning of a block, but not its end, or vice versa. Expressed in another, equivalent way, the beginning and the end of the fragment to extract must be contained in the same block, and must hence be in the same scope.

Obviously, it is also not possible to extract partial statements or partial expressions. By “partial expression”, we mean for instance an operator without all of its operands (it is however possible to extract subexpressions such as one of the operands).

With the algorithm presented in Section 4.1, there is no need to care about these two preconditions. The code differentiation process will only return “well behaved” fragments that already satisfy these two preconditions. This is ensured by either the expression splitting or expression completion step of the process, described in Section 4.1.3.

If a different algorithm were used for code differentiation, that does not enforce these preconditions, they could be checked easily with the help of the AST. Namely, a fragment to extract satisfies both preconditions if it corresponds to either:

- A full subtree of the AST; this would correspond to a single expression, a single statement, or a single construct (such as a while loop of if conditional);
- Multiple full subtrees, whose root nodes are consecutive children of the same block node; this would correspond to consecutive instructions or constructs within the same block.

To actually resolve the preconditions in case they are not satisfied, the techniques of splitting or completing expressions discussed in Section 4.1.3 can be used.

6.2 Multiple Exit Paths

6.2.1 The Problem

A method cannot be extracted if execution can leave the extracted fragment in more than one location. This is typically the case if the statements to extract contain a break or return statement. Let us show an example:
void removeTo(long id, long stopId) throws Exception {
    while (!isInterrupted()) {
        // Extract from there...
        Element elt = heap.poll();
        if (elt == null)
            throw new Exception(); // Exit path #1
        if (elt.getId() == stopId)
            return; // Exit path #2
        if (elt.getId() == id)
            break; // Exit path #3
        flush(elt); // Exit path #4
        // ...to there
        process(id);
    }
    process(id);
}

In this example, the execution can leave the fragment to extract at four different places:

- At the throw statement. More generally, any thrown exception can make the execution to leave the fragment;
- At the return statement;
- At the break statement;
- At the end of the fragment, after the last instruction, in case none of the above cases occurred.

The different exit paths are also named the outgoing control flows.

A naive cut-and-paste-like method extraction obviously produces a result that is flawed:

void removeTo(long id, long stopId) throws Exception {
    while (!isInterrupted()) {
        extracted(id, stopId);
    }
    process(id);
}

void extracted(long id, long stopId) throws Exception {
    Element elt = heap.poll();
    if (elt == null)
        throw new Exception(); // Ok
    if (elt.getId() == stopId)
        return; // Wrong semantics
    if (elt.getId() == id)
        break; // Compile error
    flush(elt);
}

The above naively extracted method has two problems:

- The return statement, although it does not produce a compile error, behaves differently than in the original method. In the original method, the return statement leaves the removeTo method,
and the “process(id)” statement never gets executed. In the transformed code, the return statement only leaves the extracted method; as a result, the while loop may iterate again, and the “process(id)” will be executed in case the loop terminates. In a correct method extraction process, the return statement would require escaping two methods at once, which is not directly possible in Java.

• The break statement produces a compile error, because the corresponding while loop is no longer there.

There is an interesting point though: thrown exceptions are not a problem. Indeed, unlike a return statement, an exception can escape more than one method. In general, a thrown exception can escape any number of methods: the underlying logic (as defined by the JVM) is that an exception escapes the methods of the stack one after another until the first method actually catching it.

In this example, the thrown exception is not a problem and the semantics are preserved. They would even be preserved if the exception were caught by the original method: it suffices to declare the exception as thrown be the extracted method if it does not include the catch block, and as not thrown if it includes the catch block. In general, exceptions are never a problem for method extraction. This is very fortunate, as we saw in Section 5.4.4 that analyzing the control flow of exceptions is very difficult. Here, exceptions can just be safely ignored.

6.2.2 A Possible Solution

Regarding the break and return statements, a possible solution has been proposed by authors for the C++ language [51], and can be summarized as follows:

• A new local variable (we will call it exitType) is created and its value is returned by the extracted method. The goal of this variable is to tell the calling method “how” the execution left the extracted method.

• All instructions that leave the extracted fragment (apart from exceptions, which are not a problem as discussed previously) have to be turned into a return statement with the corresponding value for the exitType variable.

• Code is added in the calling method just after the invocation of the extracted method. This code checks the value of the returned exitType variable, and performs the corresponding control flow break.

The following excerpt shows how this strategy works with the previous example:

```java
void removeTo(long id, long stopId) throws Exception {
    while(!isInterrupted()) {
        int exitType = extracted(id, stopId);
        if (exitType == EXIT_RETURN)
            return;
        else if (exitType == EXIT_BREAK)
            break;
        // when exitType == EXIT_NORMAL, there is nothing to do
    }
    process(id);
}
```

2Note that this example is artificially small, in order to illustrate only the key ideas. As a result there are no real reduction of the amount of code. In practice the method should be significantly larger to be worth extracting.
```java
int extracted(long id, long stopId) throws Exception {
    Element elt = heap.poll();
    if (elt == null)
        throw new Exception();
    if (elt.getId() == stopId)
        return EXIT_RETURN;
    if (elt.getId() == id)
        return EXIT_BREAK;
    flush(elt);
    return EXIT_NORMAL;
}
```

The above example shows why the strategy actually works: the extracted method no longer attempts to perform the control flow break as it was present in the initial code (either the exit from the `removeTo` method, or the escape from the `while` loop). Instead, it just passes the kind of flow break that should be issued as a result of the method. Then, the actual flow break is performed by the calling method, by looking at the returned value. Indeed, only the calling method is in a scope that matches the initial scope of the fragment to extract. Therefore only the calling method can actually perform the actual `break` statement or the correct `return` statement.

In a refactoring algorithm, this strategy can be implemented in three steps, in order to avoid adding too much complexity to the method extraction process:

- In a first step, the `exitType` variable is introduced, and the corresponding modifications are done:
  - The variable is initialized with a special value meaning “undefined”; this statement is inserted at the beginning of the fragment to extract, and becomes part of it;
  - All `return` and `break` statements within the code fragment to extract are replaced by assignments of this variable;
  - All statements after the first assignment to the `exitType` variable are surrounded by conditionals that tests for the “undefined” value. The first conditional ends at the next assignment of `exitType`, and the next one starts just afterwards, and so on. If loops of conditionals are present, the conditionals may need to be further decomposed.
  - Code handling the value of this variable, and executing the real `return` or `break` statements is inserted just after the code fragment to extract.

- In a second step only, the method is extracted. If the first step is correctly implemented, this second step works just as a regular method extraction. It does no have to deal in any way with control flow breaks.

- Finally, the `exitType` variable is removed from the extracted method by replacing its assignments by `return` statements.

In other words, the method extraction process itself is not modified. Instead, additional steps are added before and after it. The first step introduces several additional statements; however, most of them are then deleted by the third step.

The following listing shows the code of the previous example just after the first step:

```java
void removeTo(long id, long stopId) throws Exception {
    while(!isInterrupted()) {
        // Extract from there...
```
int exitType = EXIT_UNDEFINED;
Element elt = heap.poll();
if (elt == null)
    throw new Exception();
if (elt.getId() == stopId)
    exitType = EXIT_RETURN;
if (exitType == EXIT_UNDEFINED) {
    if (elt.getId() == id)
        exitType = EXIT_BREAK;
}
if (exitType == EXIT_UNDEFINED) {
    flush(elt);
    exitType = EXIT_NORMAL;
}
// ...to there
if (exitType == EXIT_RETURN)
    return;
else if (exitType == EXIT_BREAK)
    break;
// when exitType == EXIT_NORMAL, there is nothing to do
}
process(id);
}

In the second step, a regular method extraction on this modified code is performed. It produces a correct result, although a little complex compared to the desired one:

```java
int extracted(long id, long stopId) throws Exception {
    int exitType = EXIT_UNDEFINED;
    Element elt = heap.poll();
    if (elt == null)
        throw new Exception();
    if (elt.getId() == stopId)
        exitType = EXIT_RETURN;
    if (exitType == EXIT_UNDEFINED) {
        if (elt.getId() == id)
            exitType = EXIT_BREAK;
    }
    if (exitType == EXIT_UNDEFINED) {
        flush(elt);
        exitType = EXIT_NORMAL;
    }
    return exitType;
}
```

The role of the third step is to simplify the extracted method by:

- Replacing assignments of the `exitType` variable by direct `return` statement;
- Removing the introduced conditionals testing for the `EXIT_UNDEFINED` value of `exitType`;
- Removing the declaration of `exitType`. 
The final result is then the one given at the beginning of this subsection.

Regarding the third step, observe that for arbitrary code, it is not possible to replace assignments by return statements or to remove conditionals without affecting the semantics. However in this particular case the two constructs are “canceling” each other by construction, which is why they can both be removed safely. In the actual implementation, the AST nodes corresponding to the “artificial” constructs that are introduced in the first step can be “flagged” (most AST implementations allow arbitrary properties to be added to the AST nodes) so that the third step can immediately identify them.

6.2.3 Theoretical Foundations

We have presented a way of resolving the problem of multiple exit paths. While this solution was initially presented as a sort of “trick” [51], we want to present some theoretical foundations that are hidden behind it.

The problem and the proposed solution are closely related to data and control flows. Let us look at the different flows (data and control) that are present at the end of the fragment to extract (the outgoing flows), before and after the first transformation.

In the original method, there are no data flows, but three different control flows: one corresponding to the return statement, one corresponding to the break statement, and one corresponding to the normal execution. Each of them lands at a different place of the code.

After the first transformation, the situation changes. There is only one control flow left, corresponding to the normal execution. However, a data flow has been introduced, namely the exitType variable. Note that the three original control flows still exist, but they have been moved after the end of the fragment to extract.

Why can the modified code be extracted and not the original one? The reason can be summarized by the following two rules:

• the fragment to extract can have only one outgoing control flow, and it must correspond to a “normal” execution, i.e. the execution must leave after the last instruction of the fragment;

• the fragment to extract can have at most one outgoing data flow, which is the optional value that will be returned by the extracted method.

In the initial version of the code, the first rule was not satisfied, as there were three outgoing control flows. In the modified version though, there is one outgoing data flow and one outgoing control flow, and both rules are satisfied.

In a FOOD or Program Dependence Graph (graph representing both data and control flows), observe that a control flow is similar to a control dependence edge, which is itself closely related to a Boolean data flow. While the three things are nevertheless different, existing constructs based on Boolean values can be used to explain the first transformation of the code in a more formal way.

Figure 6.2 shows the FOOD graph that is equivalent to the following statements that have been added in the first step of the transformation:

```java
if (...) ... 
exitType = EXIT_RETURN;
...
if (...) ... 
exitType = EXIT_BREAK;
...
exitType = EXIT_NORMAL;
...
if (exitType == EXIT_RETURN)
```
In the FOOD graph, two constructs are of interest:

- **MUX** takes three control flows as inputs, and generates an integer data flow as output. The integer value corresponds to the index (between 0 and 2) of the control flow that is triggered (assuming exactly one of them is triggered).

- **DEMUX** takes the integer data flow as input, and generates three control flows as output. Exactly one of the output control flows is triggered, based on the value of the integer data flow (assumed to be between 0 and 2).

In other words, the transformation of the code that was applied consists in converting several mutually exclusive control flows into a single data flow, and vice versa.

In the FOOD model, these conversions are directly expressed by the MUX and DEMUX nodes. The Java implementation of these nodes basically corresponds to the statements added in the first step of the transformation. The different assignments of the `exitType` variable at different locations of the code correspond to the MUX node. Then, the if block that tests for each possible value of the `exitType` variable and performs the corresponding `break` or `return` statement corresponds to the DEMUX node.

It is interesting that exceptions do not participate in this transformation process. This does not mean that exceptions do not correspond to control flows, but rather that in the particular case of method extraction, there is no need to deal with them explicitly.
6.3 Multiple Results

In the previous section we discussed the problem of multiple outgoing control flows and proposed an effective solution. This section deals with another problem: multiple outgoing data flows. Indeed, a method cannot return more than one value in Java. Also note that in the presence of multiple outgoing control flows, the solution proposed in the previous section introduces an additional outgoing data flow. The short limit of the Java language of at most one outgoing data flow makes it important to be able to cope with situations in which a method would have to return more than one result.

In a first part, we summarize existing approaches and analyze their advantages and drawbacks, in particular with respect to the problem of forming a template method. In a second part, we propose a new approach that is specifically suited to the problem of forming a template method. We then show how the technique can be combined with the resolution of multiple outgoing control flows presented in the previous section.

6.3.1 Existing Approaches

Several approaches have been proposed in order to solve the problem of multiple results, each with its own advantages and drawbacks [24, 27, 43, 47, 51]:

- Return an object whose fields are set to the different results
- “Introduce Parameter Object”: A refactoring that encapsulates arguments and results in an object
- Create a “Method Object”
- Return an array or a collection
- Use container objects to simulate by-reference arguments
- Convert local variables passed as arguments into instance variables
- Extend or shrink the fragment to extract
- Create multiple slices
- Transform the code before the extraction

We now briefly review these approaches on the following simple example:

```java
void printResult(double x, double y) {
    double min = x - y / 2.0;  // extract from here ...
    y = y * 2.0;
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    max = max + 1.0;           // ... to here
    double result = doStuff(min, middle, max, x, y);
    System.out.println(result);
}
```

Observe that the extracted method would need to return four results: min, max, middle and y.
Returning an Object

A first solution, returning an object, is an elegant one as it perfectly fits within the spirit of object oriented languages such as Java. However, it requires the creation of a new class for each combination of returned result types. This solution is relevant when all the returned values are semantically “bound” together. For example, if a method has to return a string corresponding to some text and a Color object corresponding to the color of the text, it makes sense to create a new class (named for instance ColoredText) that encapsulates the two results. Then the method can return an instance of this class as the only result. However, there are many situations in which the method needs to return several values that are not directly related, and creating a new class to group them can be artificial, and may reduce the clarity of the code (which is obviously against the primary goal of refactoring). Additionally, the sole creation of new classes for returning multiple results increases the amount of code, and hence potentially decreases its clarity (recall Section 1.1.3).

This solution would lead to the following code when applied on the previous example:

```java
void printResult(double x, double y) {
    ResultObj o = extracted(x, y);
    double result = doStuff(o.min, o.middle, o.max, x, o.y);
    System.out.println(result);
}

ResultObj extracted(double x, double y) {
    double min = x - y / 2.0;
    y = y * 2.0;
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    max = max + 1.0;
    return new ResultObj(min, max, middle, y);
}

class ResultObj {
    double min;
    double max;
    double middle;
    double y;

    public ResultObj(double min, double max, double middle,
                     double y) {
        this.min = min;
        this.max = max;
        this.middle = middle;
        this.y = y;
    }
}
```

Observe that the resulting code is much larger than the original one. However, this solution still makes sense in clone detection and removal tools, because although it increases the amount of code, it can reduce the number of duplications if the extracted code occurs at multiple places. Furthermore, when faced to unknown code (as is frequently the case when doing design recovery), such a “return object” can help understanding what a method does, by showing what variables it affects.
Introduce a Parameter Object

This alternative solution is to use a well known refactoring: “Introduce a Parameter Object”. This refactoring applies when one or more methods take a lot of arguments. The idea in general is to create an object that encapsulates all the arguments. This new object is then passed as the only argument of the method. This differs from the previous solution which uses an object to return the results. Nevertheless, this solution can also be used to return multiple results. Indeed, the results can be indirectly returned by setting properties of the parameter object in the extracted method. The calling method then has to retrieve the results by getting the corresponding properties from the parameter object.

This solution again makes sense if all arguments and results are related. An example for instance would be a method that takes two coordinates as input and produces two coordinates as output. Such a method could have a single Point argument, and can use this argument for both reading the supplied coordinates and to store the result. As with the previous approach, this solution can be artificial in the presence of unrelated arguments and results.

This solution gives the following result when applied on the previous example:

```java
void printResult(double x, double y) {
    ParamObj o = new ParamObj();
    o.setX(x);
    o.setY(y);
    extracted(o);
    double result = doStuff(o.getMin(), o.getMiddle(), o.getMax(), x, o.getY());
    System.out.println(result);
}

void extracted(ParamObj o) {
    double x = o.getX();
    double y = o.getY();
    double min = x - y / 2.0;
    y = y * 2.0;
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    max = max + 1.0;
    o.setMin(min);
    o.setMax(max);
    o.setMiddle(middle);
    o.setY(y);
}
```

// Definition of the class ParamObj is left as an exercise
// to the reader. It’s a simple class with five properties:
// x, y, min, max and middle

Note that the extracted method fetches its arguments from the parameter object in a first step, and put the results in it in a last step. Alternatively, it could be possible to “inline” all accesses to the parameters and results by calls to the getters and setters.
Convert the Method to a Method Object

This is another refactoring that pushes the two previous ideas to their extreme. It is usually used when a method not only has a lot of arguments and results, but also has a huge body. This refactoring proposes to move the method into a new class whose sole purpose is to execute that method. The arguments and results of the methods are converted to fields of that class. The initial call to the method is hence converted into the following sequence of operations:

- Instantiate the class;
- Call the setters to fill the arguments;
- Invoke the moved method (it has no argument and no result);
- Retrieve the results by calling the getters.

This is illustrated by the following resulting code:

```java
void printResult(double x, double y) {
  MethodObj o = new MethodObj();
  o.setX(x);
  o.setY(y);
  o.extracted();
  double result = doStuff(o.getMin(), o.getMiddle(), o.getMax(), x, o.getY());
  System.out.println(result);
}

class MethodObj {
  private double x;
  private double y;
  private double min;
  private double max;
  private double middle;

  public MethodObj() {}

  // ... Getters and setters for all private fields ...

  public void extracted() {
    min = x - y / 2.0;
    y = y * 2.0;
    max = x + y / 2.0;
    middle = (min + max) / 2.0;
    max = max + 1.0;
  }
}
```

The calling method is similar to that of the previous solution. The only difference is that the extracted method now belongs to the created object and has no argument. The extracted method itself on the other hand is greatly simplified, because it now has direct access to the arguments and results. If the body of the method is huge, the extracted method can be further decomposed into sub methods in a seamless way.
Return an Array or a Collection

Rather than creating a new class just for the sole purpose of returning multiple results, one can just return an array or a collection object (such as a list) containing all the results. However, this approach is only really meaningful if all the results are of the same type, such as for instance three integers corresponding to a 3D coordinate. If the results have different types, the returned array of collection must be an array or collection of `Object` instances, and the caller has to explicitly perform type casts to retrieve the results. In such a situation, the maintainability of the code is decreased, which is again against the primary goal of refactoring.

In the case of our example, this is still relevant. The following code shows the result using an array:

```java
void printResult(double x, double y) {
    double[] mmmy = extracted(x, y);
    double result = doStuff(mmmy[0], mmmy[1], mmmy[2], x, mmmy[3]);
    System.out.println(result);
}

double[] extracted(double x, double y) {
    double min = x - y / 2.0;
    y = y * 2.0;
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    max = max + 1.0;
    return new double[] {min, max, middle, y};
}
```

The problem is the readability of the code, which is decreased by the use of indexes to identify the different results. While the result can be improved by using constants for these indexes, it is still not as elegant as the previous solutions involving objects. On the other hand, the result is clearly shorter.

Use Container Objects

The problem of methods returning multiple results does not exist in programming languages that allow parameters to be passed by reference. The Java language has no such facility; however, a possibility would be to simulate arguments passed by reference using “container objects”. The idea is illustrated by the following code:

```java
void printResult(double x, double y) {
    Ref<Double> yRef = new Ref<Double>(y);
    Ref<Double> minRef = new Ref<Double>();
    Ref<Double> maxRef = new Ref<Double>();
    Ref<Double> middleRef = new Ref<Double>();
    extracted(x, yRef, minRef, maxRef, middleRef);
    double result = doStuff(minRef.get(), middleRef.get(),
        maxRef.get(), x, yRef.get());
    System.out.println(result);
}

void extracted(double x, Ref<Double> yRef, Ref<Double> minRef,
    Ref<Double> maxRef, Ref<Double> middleRef) {
    minRef.set(x - yRef.get() / 2.0);
    maxRef.set(x + yRef.get() / 2.0);
    middleRef.set((x - yRef.get() / 2.0) + yRef.get());
}
```

126
yRef.set(yRef.get() * 2.0);
maxRef.set(x + yRef.get() / 2.0);
middleRef.set((minRef.get() + maxRef.get()) / 2.0);
maxRef.set(maxRef.get() + 1.0);
}

class Ref<E> {
  private E value;

  public Ref() {}

  public Ref(E value) {
    this.value = value;
  }

  public void set(E value) { this.value = value; }
  public E get() { return this.value; }
}

This solution is interesting from the theoretical point of view, as it shows that some sort of by-
reference addressing is possible in Java. Furthermore, the use of generics (since Java 1.5) allows the
container class Ref to be reused for any parameter type. However the solution is inelegant: it introduces
several lines of boilerplate code\(^3\) to create the references, and all accesses have to be replaced by getters
and setters. Observe that this solution is similar to the introduction of a parameter object, except that it
creates a different object for each parameter. Introducing a single object for all parameters is preferable.

In practice, it could make sense to use a solution that is half-way between the two, and to introduce
more than one parameter object in such a way each one contains a subset of parameters and results that
are related. In this particular example, one object could hold the x and y variables while a second object
could hold the min, max and middle variables.

**Convert Arguments and Results to Instance Variables**

As previously stated, only local variables need to be passed as arguments or returned as results when
extracting a method. This is *not* the case for instance variables (fields): indeed, the fields can be freely
accessed by both the extracted and the calling method. So an obvious solution (but also naive as we will
see) is to just convert all the local variables that need to be returned into instance variables:

```java
private double x;
private double y;
private double min;
private double max;
private double middle;

void printResult(double x_, double y_) {
  x = x_;
  y = y_;
  extracted();
```

\(^3\)This expression is commonly used by programmers to denote “verbose” code that is required, but does not really belong to
the business logic of the application. Boilerplate code in general does not directly implement any of the application’s features.
Boilerplate code is frequently the result of over engineering.
The result is simple and elegant at a first glance. However it has a major drawback: the extracted method is no longer *reentrant*.

A method is *reentrant* if it can safely be executed concurrently by multiple threads. In the original code, this was the case, because the method only manipulates local variables. Local variables are located on the *stack*, and as each thread has its own stack, there is no risk that the local variables of one thread get mixed with those of another thread. In the above modified code however, two different threads would now access the same variables, because they are now located on the *heap*, which is shared by all threads.

Furthermore, this solution would also change the semantics when applied on a method that calls itself in a recursive way. For these reasons, this option is rarely chosen in practice. It still makes sense when applied on a method that is not recursive, and is already synchronized (cannot be executed by more than one thread at a time) for other reasons.

A solution that is close to this one, but solves the problem of concurrency is the previously discussed solution that converts the method into a “Method Object”: indeed, in that case, a new object is created for every invocation of the method. Hence, every invocation can manipulate its own set of variables without affecting those of other concurrent invocations.

### Extend or Shrink the Fragment to Extract

Until now, we have proposed several ways of solving the problem of multiple results. However, in all the proposed solutions, we have taken the fragment to extract as a “hard” input that cannot be modified. In practice, there are several cases in which we can solve the problem by just expanding or shrinking the fragment to extract.

In our particular example, it suffices to expand it with the line just following it, and the precondition is almost magically solved: indeed, with this little modification there is now only a single result that needs to be returned, namely the *result* variable:

```java
void printResult(double x, double y) {
    double result = extracted(x, y);
    System.out.println(result);
}
```

```java
double extracted(double x, double y) {
    double min = x - y / 2.0;
    y = y * 2.0;
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    max = max + 1.0;
    // The following line has been added to the fragment to extract:
    return middle;
}
```
double result = doStuff(min, middle, max, x, y);
return result;
}

Observe that although the y variable is modified by the extracted code, it is not read by the calling method and hence does not need to be returned.

In the case of a refactoring tool, one usually expects the tool to exactly extract the statements that are selected by the user, and to report an error if the extraction is not possible. Hence this solution should not be used. However it may make sense to propose it to the user as an alternative in case the extraction fails.

In the case of clone detection on the other hand, this solution perfectly makes sense: we do not care about the exact statements that are extracted as long as duplications are reduced. Furthermore, some authors argue that a group of statements that modify a single variable usually consist of statements corresponding to a single “functionality” [24]. They are hence better candidates for extraction than statements that modify multiple variables. This hypothesis has been further extended to propose a different way of implementing the “extract method” refactoring. Instead of selecting the statements to extract, the user selects a target variable, and the tools automatically infer and extract the statements that affect that variable. This slicing approach to method extraction is beyond the scope of this thesis. The reader can refer to Ran Ettinger [24].

In the above example, the fragment to extract was expanded by adding one statement to it in order to allow the method extraction. In various situations, it may also be possible to shrink it. In the above example this is a less appealing solution because it has to be shrunk a lot in order to allow the extraction. More precisely, it has to be shrunk until it is reduced to the first line only. Although this is not an effective solution in this example, we show the result as it is the basis of the new method we want to propose:

```java
void printResult(double x, double y) {
    double min = extracted(x, y);
    y = y * 2.0;
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    max = max + 1.0;
    double result = doStuff(min, middle, max, x, y);
    System.out.println(result);
}

double extracted(double x, double y) {
    double min = x - y / 2.0;
    return min;
}
```

Create Multiple Slices

Instead of trying to extract the method “as is”, it makes sense to extract not one method, but several methods. While this is debatable when the “extract method” refactoring is used alone, this is perfectly valuable when forming a template method. Indeed, the process of forming a template method in general already has to extract multiple methods. Hence splitting a method that cannot be extracted “as is” into multiple ones fits the general process. Furthermore, method extraction is a subprocess of forming a template method; as a consequence the methods to extract are not chosen by the user (although user control could be possible), and there is hence some freedom in the choice of the exact methods to extract.

The technique of slicing provides a solution in the presence of a method that has to return multiple
results, by splitting it into several methods, one per result. Each method is obtained by creating a slice on the corresponding variable it has to return.

Applying slicing on each of the returned variables produces the following code:

```java
def printResult(double x, double y) {
    double min = extracted1(x, y);
    y = extracted2(y);
    double middle = extracted3(x, y);
    double max = extracted4(x, y);
    double result = doStuff(min, middle, max, x, y);
    System.out.println(result);
}
def extracted1(double x, double y) {
    double min = x - y / 2.0;
    return min;
}
def extracted2(double y) {
    y = y * 2.0;
    return y;
}
def extracted3(double x, double y) {
    double min = x - y / 2.0;
    y = y * 2.0;
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    return middle;
}
def extracted4(double x, double y) {
    y = y * 2.0;
    double max = x + y / 2.0;
    max = max + 1.0;
    return max;
}
```

The result is elegant, because the computation of each variable is isolated in its own method. The technique of slicing however has some drawbacks that prevent it from being applicable in some cases. Indeed, some code remains duplicated in the extracted methods, such as “y = y * 2.0”. While some of the remaining duplicated code can be further removed by applying various tricks, remaining code duplications are in general to be expected when extracting multiple slices (see Section 2.2.2). In this particular case, the duplicated code has no side effects. However, in cases in which the duplicated code has side effects, slicing cannot be used, because the side effects will then occur more than once, and the semantics will be altered.

**Code Transformations**

In addition to the techniques proposed above, there is a wide range of other solutions that work by trying to apply various semantics preserving transformations before the method extraction.
The possibilities in that area are very broad: any semantics preserving transformation can eventually transform a piece of code that cannot be extracted into an equivalent piece of code that can be extracted. There is unfortunately no “generic” algorithm for that purpose, and an arbitrary semantics preserving transformation will in general not magically resolve the preconditions of method extraction. Even if it does, it may eventually confuse the user if it alters the code too much, especially in the context of refactoring. Heavy alterations of the code are not a problem in the case of clone detection (and design recovery in general), because in this context the user usually has no prior knowledge of the code.

Several kinds of small transformations have been proposed though, each with its own algorithms, models and formalisms. Discussing each of them is beyond the scope of this thesis.

Among the code transformations that have been reported in the literature, and that help in resolving preconditions of method extraction in various cases, we can mention

- Changing the order of independent statements [24, 51];
- Swapping the operands of a commutative operator [5];
- Statement promotion (such as moving statements that does not depend on a loop in or out of that loop) [51];
- Control flow to data flow conversion (cf. Section 6.2.2)

6.3.2 A New Approach

Recall that our goal is to form a template method. Method extraction is only a part of the whole process. Its purpose is to extract the statements that do not match (between the two input methods) into other methods.

As the choice of the fragments to extract is not given by the user, extending, shrinking, splitting or modifying some fragments before their extraction are relevant solutions in cases an extraction is not possible because of multiple results.

First of all, recall that all the solutions presented in Section 6.3.1 can be used. However, they all have different advantages and drawbacks. Our aim is to propose a new technique that is better suited to the extraction of methods in the context of forming a template method. While this new technique performs well in various situations, the result heavily depends on the actual code. Hence in practice, a good refactoring tool should either propose several alternatives, or choose the best one according to some metrics.

The proposed technique aims at the following:

- Limit the amount of code generation. Hence we want to avoid creating new classes or generating boilerplate code to transmit arguments or results.
- Limit the amount of code transformations. Hence techniques that heavily modify the code before or after the extraction process have to be avoided.

Why these goals? Basically because they closely correspond to what a user expects from a refactoring tool: to perform the requested transformation and nothing more. While code transformations and the creation of new classes are relevant for clone detection and extraction, they are not expected from a refactoring tool. This does not mean that they should not be done when relevant, but rather that they are better done by the user rather than automatically. In practice, a refactoring tool can propose various solutions (including complex ones that perform severe modifications of the code). The new solution we want to propose can be one of them, and has the advantage of limiting the amount of code generation and transformation. A drawback on the other hand is that it does not always succeed.
The solution basically consists in splitting a problematic fragment to extract into multiple fragments, such that each of them can be extracted. As the process of forming a template method usually has to extract several fragments anyway, splitting one or more of them into additional methods has little impact on how the result looks like. Unlike the technique of extracting multiple slices, the proposed method does not introduce new duplications in the extracted methods.

**Splitting the Method**

The process used to split a fragment into multiple ones can be expressed recursively:

- Shrink the fragment (by removing statements from its end) until it can be extracted (until it would return only one variable);
- Consider the removed statements as a new fragment, and repeat the process on that fragment if it still cannot be extracted.

The process stops in two different cases:

- If a fragment needs to be shrunk until it is degenerate. In that case, the method extraction cannot be performed. This can happen for instance if the fragment starts with a loop that modifies two variables: there is no way to make the fragment extractable by shrinking it (except by degenerating it) because the loop must be entirely contained in the fragment to allow the extraction.
- As soon as all fragments can be extracted, the splitting process stops, and the generated fragments are all extracted.

Applied to the example of the beginning of Section 6.3.1, the first iteration results in the following code:

```java
void printResult(double x, double y) {
    double min = extracted1(x, y);
    y = y * 2.0; // continue extracting from there ...
    double max = x + y / 2.0;
    double middle = (min + max) / 2.0;
    max = max + 1.0; // ... to there
    double result = doStuff(min, middle, max, x, y);
    System.out.println(result);
}

double extracted1(double x, double y) {
    return x - y / 2.0;
}
```

There is still a large fragment to extract, but it has one line less than the initial one. After five iterations, the final result is achieved, and looks as follows:

```java
void printResult(double x, double y) {
    double min = extracted1(x, y);
    y = extracted2(y);
    double max = extracted3(x, y);
    double middle = extracted4(min, max);
    max = extracted5(max);
    double result = doStuff(min, middle, max, x, y);
}
```
System.out.println(result);
}

double extracted1(double x, double y) {
    return x - y / 2.0;
}

double extracted2(double y) {
    return y * 2.0;
}

double extracted3(double x, double y) {
    return x + y / 2.0;
}

double extracted4(double min, double max) {
    return (min + max) / 2.0;
}

double extracted5(double max) {
    return max + 1.0;
}

Observe that, like with the technique of slicing, we extracted several methods. Unlike slicing on the other hand, there are no remaining duplications in the result.

This is the basic process, illustrated on a single method. When dealing with the “form template method” refactoring, we have to apply it to two methods at once. The adaptation of the algorithm from one method to two is not straightforward, because the two fragments to extract are, by definition, different. This raises the following problems:

- How to ensure that both fragments are split into the same number of sub fragments;
- How to ensure that the corresponding extracted methods have the same signatures in both cases.

These two issues have to be solved. Indeed, the template method must be unique; hence, the number of extracted methods must match in the two original methods and all methods must have the same signature.

**Working on two Methods**

Now, let us extend the example of the previous section into a real-world one, where we need to extract two different methods at the same time. Assume that the two code fragments are the following:

```java
/* First fragment */
min = x - y / 2;  // extract from here...
y = y * 2;
max = x + y / 2;
middle = (min + max) / 2;
max = max + 1.0;
min = min + 1.0;  // ...to here
doStuff(min, middle, max, x, y);

/* Second fragment */
```
min = x + 2; // extract from here...
middle = x * y;
max = min + middle;
min = min + 1.0;
y = x + min; // ...to here
doStuff(min, middle, max, x, y);

If we proceed like with the case of a single method, we can manage to get the first step correctly. Indeed, in both fragments, we need to extract a method that returns the same variable, min. The arguments are different: the first fragment requires both the x and y variables, whereas the second one only requires x. However, we can easily take the union of the required arguments to get the arguments of the final method. Its signature is given by:

```cpp
int extracted1(int x, int y);
```

The corresponding call in the two original methods (and of the future template method) is:

```cpp
min = extracted1(x, y);
```

The problems begin in the second step. Indeed, the first fragment modifies the y variable, whereas the second fragment modifies the middle variable. The first fragment would need to extract a method that returns y while the second one requires a method that returns middle. The solution we propose is to extract two methods, one for each variable, such that the future template method will look like:

```cpp
min = extracted1(x, y); // This was the 1st step
y = extracted2a(y); // 2nd step, 1st extracted method
middle = extracted2b(middle, x, y); // 2nd step, 2nd extracted method ...
```

How are the methods extracted2a and extracted2b implemented? In the first class, extracted2a obviously contains the extracted expression that is assigned to y:

```cpp
/* In the first class: */
int extracted2a(int y) {
    return y * 2;
}
```

And in the second class? As the second fragment does not modify y at this stage, the extracted2a method hence has an empty implementation:

```cpp
/* In the second class: */
int extracted2a(int y) {
    return y;
}
```

Similarly, the extracted2b method has an empty implementation is the first class because the first fragment does not modify the middle variable. Only the implementation is the second class is not empty:

```cpp
/* In the first class: */
int extracted2b(int middle, int x, int y) {
    return middle;
}
```

```cpp
/* In the second class: */
int extracted2b(int middle, int x, int y) {
    return x * y;
}
```
There is just a small trick to notice: although the second fragment only uses the variables $x$ and $y$ to set the value of the `middle` variable, the `extracted2b` method additionally needs to have the `middle` variable as argument. The reason is precisely to allow the empty implementation of the first class to return the `middle` variable unmodified.

To summarize what we did in the two first steps, we basically handled two different situations:

- In case 1, the two fragments modify the same variable. A single method can be extracted, that returns the value for that variable.
- In case 2, the two fragments modify two different variables. Two methods are extracted, one for each variable. For each method, one of the two implementations is empty.

These two situations suffice to handle most of the remaining fragments. Continuing with the rest of the two fragments to extract, we get the following result for the template method:

```java
min = extracted1(x, y); // Case 1
y = extracted2a(y); // Case 2
middle = extracted2b(x, y);
max = extracted3(x, y, min, middle); // Case 1
middle = extracted4a(middle, min, max); // Case 2
min = extracted4b(min);
max = extracted5a(max); // Case 2
y = extracted5b(y, x, min);
min = extracted6(min); // Case 3
```

Only the last line deserves some attention. In this case (Case 3), the second fragment modifies `min` whereas the first fragment basically does nothing, because its end has been reached. Hence a single method can be extracted. Its implementation returns the “$min + 1$” expression in the second class, and is empty in the first one.

Each time we are in the case 2, we have to produce two empty implementations out of four. Whenever we are in the case 3, we have one empty implementation out of two. It is clear that Case 1 results in the best code, as it permits a single method to be extracted, with no empty implementation. Case 2 is obviously the worst case, with four methods and two empty implementations.

In summary, we were successfully able to extract two completely different methods, by splitting it into 9 sub-methods. However, out of the 18 implementations of these methods (each method has two implementations, one in each class), 7 implementations are empty (6 for the three occurrences of case 2, one for the single occurrence of case 3), meaning almost half of them!

Fortunately, there are possibilities to improve this result. First, let us consider what we did in a formal way. Consider the two vectors given by the successive variables that are modified in the first and second fragment. They are given by:

```
[ min, y, max, middle, max, min ]
[ min, middle, max, min, y ]
```

Observe that case 1 has occurred exactly when, for a given index, the two vectors have the same value. In our particular case, this occurred at indexes 1 and 3 (counting from 1), were both vectors contain the same variable, `min` and `max`. Also notice that we put an empty entry at the end of the second vector, to match the size of the first one. The situation in which one of the two vectors has an empty entry corresponds to the case 3.

Now imagine that we are allowed to introduce any number of empty elements in either vectors, and consider the following modified vectors:
There are now four indexes at which the vectors have the same value! A quick calculation reveals that we would have only three empty implementations. Indeed, the template method with these modified vectors would be:

\[
\begin{align*}
\text{min} &= \text{extracted1}(x, y); \quad \text{// Case 1} \\
y &= \text{extracted2}(y); \quad \text{// Case 3} \\
\text{max} &= \text{extracted3}(\text{max}, x, y); \quad \text{// Case 3} \\
\text{middle} &= \text{extracted4}(\text{min}, \text{max}, x, y); \quad \text{// Case 1} \\
\text{max} &= \text{extracted5}(\text{max}, \text{min}, \text{middle}); \quad \text{// Case 1} \\
\text{min} &= \text{extracted6}(\text{min}); \quad \text{// Case 1} \\
y &= \text{extracted7}(y, x, \text{min}); \quad \text{// Case 3}
\end{align*}
\]

How did the introduction of empty elements in the two vectors affect our algorithm? Basically, it changed the way we associate lines of codes between the two fragments. In the initial version, we just matched line 1 with line 1, line 2 with line 2, and so on. Line 6 of the second fragment was unmatched and yielded case 3. By introducing empty elements, we changed the way lines are matched: line 1 is still matched with line 1, but then lines 2 and 3 of the first fragment are unmatched; then, line 4 of the first fragment is matched with line 2 of the second one. The following illustrates the new matching by placing the two fragments on two columns:

```c
/* First fragment */ /* Second fragment */
min = x - y / 2;      min = x + 2;
y = y * 2;            
max = x + y / 2;      middle = x * y;
middle = (min + max) / 2;    middle = min + middle;
min = min + 1.0;      min = min + 1.0;
max = max + 1.0;      y = x + min;
```

The remaining question is how can we, given two vectors corresponding to the modified variables, compute an optimal alignment, that maximizes the number of matched indexes.

The answer is simple: this is again an occurrence of the Longest Common Subsequence (LCS) problem!

This problem was already presented in Section 4.2 and is hence not covered again.

Notice that in Section 4.2 it was used for the differentiation of two method bodies. Now it is used for a different problem: to maximize "matches" between two lists of modified variables.

Observe that with this approach, it is desirable to merge consecutive statements that modify the same variable: these statements can obviously be extracted into the same method. Hence, just after the write-access vectors are built, consecutive occurrences of the same token must be replaced by a single occurrence. This did not occur with the above example, as no consecutive instructions modify the same variable.

Like for the nodes of an AST that maintain the corresponding positions in the source code, the elements of the vectors must obviously contain links to the corresponding AST nodes, so that the extraction can be performed after the different methods have been determined.

In some cases, it is impossible to split the method into multiple methods with the proposed approach. We already mentioned the case in which a loop is present. The same situation occurs in the more general case in which we are in the presence of a block (body of a loop, conditional, try, catch, etc) containing assignments to two or more variables that need to be returned.
Another situation that deserves some attention is when a single statement modifies more than one variable. This may occur, for instance, in the presence of multiple “++” or “--” prefix or postfix operators within an expression. The problems lies in the fact that whenever a method that returns a result is extracted, the original code is replaced by an assignment, with the modified variable on the left hand side and the method invocation on the right hand side. In the presence of more than one “++” or “--” operator nested in an expression, it would imply that the operator is then replaced by an assignment nested in an expression. While putting an assignment within an expression is allowed, it is quite dirty style.

Choosing Split Points

We have shown that if we have to extract a pair of methods that need to return more than one value, it is sometimes possible to split the methods into more than one pair. We also showed how to do the splitting, namely by gathering the write accesses into vectors, and applying the LCS algorithm on the two produced vectors.

In the general cases though, not all statements do perform write accesses. Hence an ambiguity remains on where to “cut” the code between two different methods. As an example, consider the following code:

```java
/* First method */
1  x = x - z / 2; // extract from here...
2  System.out.println(x);
3  System.out.println(y);
4  y = y * 2; // ...to there
5  doSomething(x, y);
```

This example is similar to the begin of the previously discussed one, except that two lines have been inserted between the two write accesses. These two additional lines do not make any write access, and hence do not need to be extracted into their own methods. Instead, they can just be included into either the first or the second method; but which one is the best? There is in fact a third choice: extracting Line 2 with Line 1 and extracting Line 3 with Line 4. Actually, all the three solutions are valid, in the sense they will not prevent the methods from being extracted (they may just require new arguments to be added to the extracted method).

A reasonable choice is to “cut” the code at the location that will minimize the number of additional arguments for the two methods. For instance, cutting between Lines 3 and 4 would require the first method to have `x`, `z` and `y` as argument and the second method to have only `y`. On the other hand, cutting between Lines 2 and 3 would require `x` and `z` for the first method, and `y` for the second. This is globally one less argument and is hence more appealing. The reader can verify that cutting between Lines 1 and 2 requires one more argument than cutting between Lines 2 and 3.

Hence, cutting between Lines 2 and 3 can be considered as the best choice. Whether it really makes sense from a semantic point of view is another question, and is certainly debatable in many cases. However, grouping statements that affect the same variables has been shown in other areas (such as slicing [24]) to tend to group statements that are semantically related.

Finally note that there are many situations in which multiple different cutting points can lead to the same number of arguments. There is no final answer to the question of which one is better in such a case. In practice, in such a situation a concrete tool may want to either:

- Choose a cut point among the best ones based on any other decision, such as
  - The first or the last cutting point;

---

4 In Java an assignment can occur within an expression, and the “value” of the assignment is the assigned value itself. However this is a “legacy” practice that is highly unrecommended and originates from the C language. Many Java development environments such as Eclipse issue a warning when an assignment is used as an expression.
The cutting point that makes the length of the extracted methods the closest in size.

- Ask the user for the best choice. In that case, it may also make sense to allow the user to choose some suboptimal cutting points as they may be more relevant according to the semantics.

Generalizing the choice of the cutting point for a pair of methods is a little harder, however just choosing the best cutting point separately on the two methods has shown to give fair results in practice.

### 6.3.3 Combining with Multiple Exit Paths

We now discuss how the problems of multiple exit paths and of multiple results can be handled together.

Section 6.2 has dealt with the presence of control flow breaks within code fragments to extract, such as the presence of `break` or `return` statements. The proposed solution required the introduction of an additional result to be returned by the extracted method. As a consequence, the limit of one result is quickly exceeded.

The solution that was proposed in Section 6.2.2 can be combined with the method proposed in Section 6.3.2. Recall that it was proposed to implement the of Section 6.2.2 in two different steps: in a first step, the exit paths are converted into a single data flow by introducing an `exitType` variable. In a second step only, the method is actually extracted. The motivation of this separation was that only the first step is specific to the problem of multiple exit paths, whereas the second step is just reduced to a standard method extraction.

The benefits of such an approach are now even clearer: to combine it with the approach discussed in Section 6.3.2 in order to resolve multiple results, there is no special care to take. It suffices to extend the second step, the “regular” method extraction, with what has been presented in the last section.

The following code illustrates the process:

```java
void method1(long id, long stopId) throws Exception {
    while (!isInterrupted()) {
        Element elt = heap.poll();
        // Extract from there...
        if (elt == null)
            throw new Exception();
        if (elt.getId() == stopId)
            return;
        elt = convert(elt);
        // ...to there
        store(elt);
    }
    process(id);
}

void method2(long id) throws Exception {
    while (!isInterrupted()) {
        Element elt = heap.poll();
        // Extract from there...
        if (elt == null)
            throw new RuntimeException();
        if (elt.getId() == id)
            break;
        flush(elt);
    }
}
```

138
This code has multiple exit paths in both methods. However, it also has to return the `elt` variable. Converting the exit paths into a variable will imply that the method will have to return two variables.

The first step is to convert the exit paths into a variable. At this stage, nothing is extracted. This results in the following code:

```java
void method1(long id, long stopId) throws Exception {
    while(!isInterrupted()) {
        Element elt = heap.poll();
        // Extract from there...
        int exitType = EXIT_UNDEFINED;
        if (elt == null)
            throw new Exception();
        if (elt.getId() == stopId)
            exitType = EXIT_RETURN;
        if (exitType == EXIT_UNDEFINED) {
            elt = convert(elt);
            exitType = EXIT_NORMAL;
        }
        // ...to there
        if (exitType == EXIT_RETURN)
            return;
        else if (exitType == EXIT_BREAK)
            break;
        store(elt);
    }
    process(id);
}

void method2(long id, long stopId) throws Exception {
    while(!isInterrupted()) {
        Element elt = heap.poll();
        // Extract from there...
        int exitType = EXIT_UNDEFINED;
        if (elt == null)
            throw new RuntimeException();
        if (elt.getId() == id)
            exitType = EXIT_BREAK;
        if (exitType == EXIT_UNDEFINED) {
            flush(elt);
            exitType = EXIT_NORMAL;
        }
        // ...to there
        if (exitType == EXIT_RETURN)
            return;
    }
    process(id);
}
```
else if (exitType == EXIT_BREAK)
    break;

store(elt);
}  

process(id);
}

The second step is the “regular” method extraction, but enhanced with the handling of multiple results as discussed in Section 6.3.2. Observe that within the code to extract, exitType is written first (at two places), then elt, and then exitType again. However, the first step has slightly changed the situation: the statement “exitType = EXIT_NORMAL” as well as the surrounding condition are now common to both methods and do not need to be extracted. Hence only two methods remain to be extracted. The result of extracting the two methods is then:

```java
void templateMethod(long id, long stopId) throws Exception {
    while(!isInterrupted()) {
        Element elt = heap.poll();
        int exitType = extracted1(elt, id, stopId);
        if (exitType == EXIT_UNDEFINED) {
            elt = extracted2(elt);
            exitType = EXIT_NORMAL;
        }
        if (exitType == EXIT_RETURN)
            return;
        else if (exitType == EXIT_BREAK)
            break;
        store(elt);
    }
    process(id);
}
```

/* In the first class */
int extracted1(Element elt, long id, long stopId) {
    int exitType = EXIT_UNDEFINED;
    if (elt == null)
        throw new Exception();
    if (elt.getId() == stopId)
        exitType = EXIT_RETURN;
    return exitType;
}

/* In the second class */
int extracted1(Element elt, long id, long stopId) {
    int exitType = EXIT_UNDEFINED;
    if (elt == null)
        throw new RuntimeException();
    if (elt.getId() == id)
        exitType = EXIT_BREAK;
}
As suggested in Section 6.2.2, a third step can then be applied to cleanup the resulting code, namely by replacing the \texttt{exitType} variable by \texttt{return} statements. The \texttt{extracted1} method can hence be simplified.

It then looks as follows:

\begin{verbatim}
/* In the first class */
int extracted1(Element elt, long id, long stopId) throws Exception {
    if (elt == null)
        throw new Exception();
    if (elt.getId() == stopId)
        return EXIT_RETURN;
    return EXIT_UNDEFINED;
}

/* In the second class */
int extracted1(Element elt, long id, long stopId) throws Exception {
    if (elt == null)
        throw new RuntimeException();
    if (elt.getId() == id)
        return EXIT_BREAK;
    return EXIT_UNDEFINED;
}
\end{verbatim}

This example shows two things: on one hand it can be possible to resolve multiple exit paths and multiple results together. On the other hand however, the result can quickly become more complex than the original code.

Because the second step is not more than a “regular” method extraction, and does not have to deal with the multiple exit paths, any of the existing techniques presented in Section 6.3.1 can also be used to resolve the multiple results.

\subsection*{6.3.4 In the Real World}

We have reviewed several ways of handling situations in which a method to extract has to return more than one value. We have then introduced a new approach. While this approach is not suitable for method extraction when used alone (because it splits the method extraction into multiple ones), it is relevant in the context of forming a template method, which is expected to require many method extractions anyway.

Among the different ways of resolving multiple results, none of the approaches can be classified as
“the best” one in practice. Indeed, for each approach, one can find both a code sample for which it gives the optimal result and another code sample for which it gives the worst result.

The new approach presented in Section 6.3.2 works best when the fragment to extract makes clearly distinct tasks one after the other. On example is the following code. One method first computes and displays the sum of an array of elements, and then computes and displays the product of the same array. The second method does the same, except that it adds some computation on the array values:

```java
public void method1(double[] arr) {
    // extract from here
    double sum = 0.0;
    for (int i = 0; i < arr.length; i++)
        sum += arr[i];
    System.out.println("Sum:" + sum);
    double prod = 1.0;
    for (int i = 0; i < arr.length; i++)
        prod *= arr[i];
    System.out.println("Product:" + prod);
    // ...to there
    ... some more similar code that uses sum and prod
}

public void method2(double[] arr) {
    // extract from here
    double sum = getBaseSum();
    for (int i = 0; i < arr.length; i++)
        sum += 1.0 / arr[i];
    System.out.println("Sum:" + sum);
    double prod = getBaseProd();
    for (int i = 0; i < arr.length; i++)
        prod *= arr[i] + 1.0;
    System.out.println("Prod:" + prod);
    // ...to there
    ... some more similar code that uses sum and prod
}
```

The code to extract modifies two variables, `sum` and `prod`. Hence the proposed technique will extract two methods instead of one. The result consists of a clean pattern, in which the first extracted method computes and displays the sum, and the second one computes and displays the product (only the extracted methods of the first class are shown):

```java
public void template(double[] arr) {
    double sum = extracted1(arr);
    double prod = extracted2(arr);
    ... some more similar code that uses sum and prod
}
```

Observe that some similarities remain between the two fragments to extract. Recall however that it is relevant to introduce a threshold in the differentiation process to avoid the extraction of methods that are too small. Hence the presence of remaining similarities is not a mistake, and does not prevent the template method from being created.

More generally, a special version of the “form template method” refactoring might drop the differentiation step altogether, and always attempt to extract the whole body. This would produce a template method that only consists of invocations of delegate methods, one per variable modification. The result would then be closely related to the alternative definition of the refactoring discussed in Section 3.1.
public void method1(double[] arr) {
    // extract from here...
    double sum = 0.0;
    double prod = 1.0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
        prod *= arr[i];
    }
    System.out.println("Sum: " + sum);
    System.out.println("Product: " + prod);
    // ...to there
}

public void method1(double[] arr) {
    // extract from here...
    double sum = 0.0;
    double prod = 1.0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
        prod *= arr[i];
    }
    System.out.println("Sum: " + sum);
    System.out.println("Product: " + prod);
    // ...to there
}

This result does not involve returning an array, collection or object; or creating and making use of containers or method objects, and is hence simpler than all the other discussed approaches.

On the other hand, the same code can become a difficult case by just moving the statements (only the first method is shown; assume that the second method is arranged in the same way):

public void method1(double[] arr) {
    // extract from here...
    double sum = 0.0;
    double prod = 1.0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
        prod *= arr[i];
    }
    System.out.println("Sum: " + sum);
    System.out.println("Product: " + prod);
    // ...to there
}

The resulting template then becomes:

public void template(double[] arr) {
    double sum = extracted1(sum);
    double prod = extracted2(prod);
    for (int i = 0; i < arr.length; i++) {
        sum = extracted3(sum, arr, i);
        prod = extracted4(prod, arr, i);
    }
    extracted5(sum, prod);
}

The modified code now requires the extraction of five method pairs. In that case, returning an array of two elements, or extracting two slices are both much better solutions.

In general, when implementing the form template method refactoring, one may eventually think about testing many approaches, and choosing the best one according to some metrics on the resulting code (number of lines, etc).
However, the solution that will probably be chosen by the user in presence of the second version of the code is to first convert it to the first version in which \texttt{sum} and \texttt{product} are computed one after the other (this can be done using slicing), and to form a template method in a second step only.

In short, there is no simple way of determining the optimal choice. For that reason, interaction with the user is an essential concern for an automated refactoring tool. While the tool may show to the user the results of applying different approaches, canceling the refactoring is also an option that must be proposed. Indeed, \textit{no} refactoring is useful in \textit{all} situations. Forming a template method works fine when:

- Two methods have a similar bodies;
- The two methods do not have too much interleaved write accesses to different local variables.

If any of these two conditions is not fulfilled, the refactoring may either fail, or produce a result that is more complex than the original one (although still semantically equivalent to the original, which is our goal). This is not new and applies to any refactoring: overuse of refactoring in unnecessary situations can quickly result in over engineering, as discussed in Section 1.1.3. As already mentioned, we assume that a refactoring tool only has to produce a correct result; the decision of what refactoring to apply where and when is another problem.

\section{Summary}

This chapter has explored a problem that is already known in the field of clone detection and extraction: what can be done when statements cannot be extracted because a precondition of the “extract method” refactoring is not fulfilled? We gave an extensive overview of the various existing approaches that have been explored in the area of clone extraction and that can potentially be reused in the context of refactoring. However, we showed that although the two problems share similarities, they still have different goals. Therefore we proposed an additional approach that is better adapted to the context of refactoring, and briefly highlighted how it can fit together with the other approaches on real-world situations.
Chapter 7

Related Work

This chapter is devoted to a brief exploration of some areas of research that are different, but share some similarities with the implementation of automated refactoring.

Refactoring belongs to the broader class of code transformations, and other kinds of transformations obviously share some of the underlying techniques. Section 7.1 explores some of them: compiler optimization, profiling and SQL manipulations. Then a subsection explores refactoring in a context that is broader than just Java code, and includes several other artifacts (JSP pages, XML files, etc).

Refactorings can also be viewed as \textit{model transformations}, where the “model” is a suitable representation of the code to transform, such as the Class Graph or the AST. Models are not only useful for transformations, but also for design, documentation and security. This is explored in Section 7.2.

7.1 Analyses and Transformations

Algorithms and applications that analyze and transform the code automatically have existed long before the introduction of refactoring tools. One of the oldest tool that analyzes and transforms code is the compiler.

7.1.1 Compiler and Compiler Optimizations

Whereas refactoring transforms source code into source code, a compiler transforms source code into byte code or assembly code. A refactoring is hence a transformation from one language to the same language, whereas a compiler transforms one language into another. The target language of a compiler does not even need to be assembly or byte code: various compilers from one language to another exist, such as Modula-2 to C\textsuperscript{1}, Pascal to Java\textsuperscript{2}, UML to Java\textsuperscript{3} and Java to JavaScript\textsuperscript{9}.

Compiler and refactoring tools share many techniques. Both of them for instance make use of the AST and Class Graph. The techniques are however used for purposes that can be different. Only compilers for instance use the AST to check for errors; refactoring tools usually assume correct code as input. Conversely, both of them need to use origin tracking (see Section 2.2.4); however, a compiler uses it to link compile errors to the corresponding source code locations whereas a refactoring tool uses it to actually transform the code without resorting to a full rewriting of the AST.

The field of compiler optimization shows even more similarities with refactoring. Indeed, many optimizations are closely related to specific refactoring or refactoring processes [28, 79]:

\begin{itemize}
\end{itemize}
• Data flow analysis is used by compilers to optimize the use of CPU’s registers [32, 48]. In refactoring, this is used to minimize the number of arguments and results in an “extract method” refactoring.

• Compilers can try to remove redundant code in several ways in order to reduce the code size. The used methods closely correspond to specific refactorings: extract method, extract local variable.

• On the other hand, several aspects of the code can be flattened by a compiler, either because they cannot be expressed in the target language (assembly or byte code), or because they result in faster code. These operations again correspond to well-known refactorings: in-line method, in-line constant, in-line temp, reverse conditional, unroll loops, remove double negatives, etc.

Nevertheless, there is a major difference between compilers and refactoring tools: their purpose. The goal of refactoring is to improve the design of code, or its readability. A compiler on the other hand, has to produce code that is as fast as possible. Performance and readability are usually antagonistic goals.

The fact they share many techniques can be understood if we recall that badly used refactorings can produce code that is more difficult to read rather than better designed code. More generally, refactoring can indeed sometimes be used on purpose for goals that are different than design improvement. In particular, some refactorings have been proposed to primarily address performance issues rather than design improvements [1, 47].

7.1.2 Profiling and Debugging

Memory leaks, null pointers, buffer overruns, segmentation faults; these are all nightmares for any programmer who has experienced complex code using pointers on a programming language without type safety (such as C or C++). Advanced profiling and debugging tools aim in detecting such bugs.

While some bugs, such as null pointers, can be simply detected through the insertion of additional verification code by the compiler, other bugs such as memory leaks or segmentation faults would be too heavy (and introduce a large speed penalty) to detect in a similar way, and verification code is usually not inserted by compilers. However, advanced tools are able to detect such bugs.

Because the compiler does not produce the necessary verification code, such tools have to rely on other approaches. One of them is to transform the source code or the object code, or in case of Java, the byte code [56]. This approach is taken, for instance, by the Purify tool of IBM Rational. However, in most cases, the transformations are limited to the addition of verification code, and hence differs from compilation, optimization or refactoring.

Another approach to profiling and debugging is to emulate the CPU. Indeed, through emulation, each CPU instruction can potentially be augmented by performing additional tests before or after its execution. Any dangerous instruction (such as an illegal memory access) is intercepted by the emulator and is canceled and reported. This is the approach taken, for instance, by the Valgrind open source tool.

For a detailed survey of profiling and debugging tools, refer to Glenn R. Luecke et al. [56].

Both approaches are to some extent comparable to the decorator and wrapper design patterns. Indeed, they both “augment” individual instructions with sanity checks. They are hence comparable to the “create wrapper class” refactoring proposed by some IDEs. On the other hand, the transformation is usually performed by inserting the code at every required place rather than through a clean design pattern, mainly for performance reasons, and because the programmer never has to deal with the transformed code directly anyway.
7.1.3 SQL Query Processing and Transformations

Like Java, SQL is a language with a well defined syntax, and an AST can be built from an SQL statement. The SQL language is surprisingly more powerful than what might be expected from a beginner’s book on the topic. For instance, SQL “SELECT” statements are usually presented as follows in beginner books:

```
SELECT <column1>,<column2>,... FROM <table>
```

However, a more correct (but still largely incomplete) representation, although more confusing at a first glance, is:

```
SELECT <expr1>,<expr2>,... FROM <matrix_expr>
```

Here, “<expr>” denotes any expression that return a single value, whereas “<matrix_expr>” denotes any expression that returns a matrix (a bi-dimensional array of values that can be viewed as a table).

The simplest expression that return a single value is a column of the table, but expressions involving operators are also permitted (such as “SELECT column1 + column2, 3 * column3,...”). Even a full SELECT statement can return a single value and can hence be used as an expression within another SELECT statement: “SELECT column1, (SELECT MAX(column) FROM ...), column3 FROM table” is legal.

The simple kind of matrix expression is a table. However, a SELECT statement itself can return a matrix and is hence allowed in the “FROM” clause of another SELECT statement. “JOIN” and its derivatives (inner, left, right and outer joins) are just operators that take two matrix expressions and one Boolean expression as arguments, and produce a new matrix expression. Statements such as “SELECT ... FROM ((SELECT ...) INNER JOIN table1 ON ...) LEFT JOIN (SELECT ...)” are perfectly legal SQL statements.

As for analyzing Java code, building an abstract representation (such an AST) is mandatory for any database that aims to optimize SQL statements and execute them in the fastest way. Statements that are very similar may require completely different execution schemes. Consider for instance the following two statements (taken from a real-world example). Assume here that we have two tables, “product” and “history”, related by a one-to-many relation (many history rows for each product row):

```
SELECT * FROM product
    WHERE product.name = 'Java'
    AND EXISTS
        (SELECT * FROM history
            WHERE history.date < now())

SELECT * FROM product
    WHERE product.name = 'Java'
    AND EXISTS
        (SELECT * FROM history
            WHERE history.date < now()
            AND history.productId = product.id)
```

It seems at a first glance that the second statement is just slightly more complex than the first one, as it barely adds an additional criterion, namely “history.productId = product.id”. There is a major difference though: in the second statement, the inner “SELECT” refers to a column of a table that is selected by the outer “SELECT”, namely “product.id”. The inner query is said to be correlated to the outer one in this case. This has a serious influence on how a database can correctly execute the two queries:
For the first statement, the inner “SELECT” can be executed first, and only once, because it does not depend on the outer one. Then in a second step, the outer “SELECT” can be executed.

For the second statement, the inner “SELECT” must be executed for every row returned by the outer “SELECT”. Indeed, it requires the value of the column “product.id”, which is obviously different for every row! As a result, the second statement may take much more time to execute. If both tables have the same size $n$, there is indeed a change of the complexity from $O(n)$ to $O(n^2)$.

Here a familiar notion emerges: the scope. “SELECT” statements act like Java blocks: columns of the outer query (columns of the “product” table are visible in that query and in all inner queries. Columns of the inner query (columns of the “history” table) on the other hand are not visible in the outer query, but would be visible in deeper inner queries. Unlike with Java, the fact a column is used in a nested query has impact on the execution. Nevertheless, the analysis of columns and their scopes in SQL is similar to the analysis of variables and their scope in Java.

This example makes it clear that a correct analysis of SQL statement is required for their proper execution.

However, the second statement of the example does not need to be slower than the first one if carefully chosen transformations are introduced. Indeed, it can be converted into the following equivalent pseudo-SQL statement, that runs in $O(n)$ time:

```
SELECT * FROM products
  LEFT SEMI JOIN history ON history.productId = product.id
  WHERE product.name = 'Java'
  AND history.date < now()
```

Here, “LEFT SEMI JOIN” is a pseudo-SQL operator that is similar to a regular “INNER JOIN”, except that it only joins the first row of the second table (“history”) that matches the “WHERE” condition and the join criteria (“history.productId = product.id”) for a given row of the first table (“product”). A regular “INNER JOIN” would join all the matching rows and eventually return the same rows from the “product” table several times.

Note that with this transformed statement, a join operation is involved, but unlike in the previous example, there is no longer a need to execute some statement multiple times. In fact, the transformed query can execute much faster, as join operations typically run in linear time when the concerned columns are indexed. Good databases automatically do such a transformation when in presence of a correlated inner query. The transformation is not always possible though, but this is beyond the scope of this section.

To summarize, analysis and transformation can be useful on various other languages, including languages that are not necessarily general purpose (ignoring vendor-specific SQL extensions). We have briefly provided illustrations using SQL as an example. This example showed both similarities and differences with Java as a target language. In the similarities, we note the need for abstraction, and hence for abstract models such as the AST. In the differences, we note the goals: proper execution and optimized execution of SQL, versus refactoring of Java. Note that optimized execution was also mentioned in the previous section regarding compilers.

The next section goes one step further in the exploration of languages that radically differ from Java.

### 7.1.4 Web Development: The Death of Refactoring and Maintainability?

In this thesis, we started from the state of the art in the field of refactoring, and we proposed new techniques to go beyond what already exists. However, our study was only considering the evolution of
refactoring techniques: how can existing techniques be improved, and how can new techniques (more complex refactoring) be implemented?

It is hence legitimate to also consider another evolution: the evolution of programming languages and frameworks. Indeed, new languages and new frameworks will obviously require refactoring tools and algorithms to evolve as well.

Here we only discuss one particular case: the recent trend toward web applications. The reason of this choice (among all the possible new programming languages, paradigms and frameworks) is that web applications and frameworks, in their current state, are big troubles for refactoring. Static analyses and automated refactoring of web applications are not only much more difficult than with regular stand-alone applications, but most existing frameworks introduce several design choices that just make even the simplest automated refactoring impossible to implement correctly. Furthermore, looking at the causes of this problem reveals a lack of abstraction within existing web frameworks compared to accepted object oriented practice.

This section is mainly based on Java Server Pages (JSP) and Java Server Faces (JSF), which are the standard frameworks proposed by Sun as a part of Java Enterprise. However, most of our comments also apply to several other existing web frameworks.

More precisely, a tool that wants to implement automated refactoring for web applications (and not just for plain Java applications) has to cope with the following additional challenges:

- Multiple different languages and syntaxes to consider (Java, HTML, XML, DTD, XSLT, HQL, SQL, EJB-QL, JavaScript, JSP pages, Expression Language (EL), XPath, property files, Maven, Ant, JavaFX, just to cite a few)
- Embedding of one language into another (Java within JSP; EL within JSP, XML or Java; JavaScript within Java or HTML; Java variable names within JSP, XML or property files; and much more...)
- Presence of languages that are difficult or impossible to refactor accurately (typically JavaScript)
- Lack of abstraction and overuse of strings.

We now review each of these issues, and briefly summarize the possible or existing solutions (if any). Our aim is not to find working solutions, but rather to present the problem and highlight future research directions.

**Multiple Languages**

From the theoretical point of view, the presence of multiple languages is not a limitation for refactoring. It suffices to implement the factorings for all the languages. This is more a practical limitation, because it obviously requires much more code to be written in order to implement a refactoring tool, especially if the different languages have few things in common.

Note that the first step for the implementation of a refactoring tool is usually to write a parser that transforms the underlying language into a more abstract representation such as the AST. Such a representation is not only useful for refactoring, but it also allows advanced features such as syntax checking, auto-completion, references searches\(^5\) and in-line help (such as pop-ups showing details of an element when moving the mouse on it).

Recent development environments supporting the development of web applications (such as Eclipse or NetBeans), are able to parse most of the different used languages (XML, HTML, EL, JavaScript). However, as of writing this thesis, the support for advanced features is very recent and not mature, despite the fact that some of the corresponding languages have existed since several years. In particular, support

\(^5\)A “references search” for instance is a feature that allows you to find all occurrences of a variable, and that properly differentiates different variables with the same name.
for refactoring, syntax checking, auto-completion, reference searches or in-line help is sometimes still limited or absent\textsuperscript{6}.

**Difficult to Refactor languages**

It is well known that some programming languages (such as C) are difficult to refactor \cite{33, 34}. There are several aspects that make the automation of refactoring more difficult, and for some refactoring and some situations impossible. By “impossible”, we mean all situations in which the correctness of the refactoring cannot be guaranteed. In several such situations though, it is still possible to make some guesses that produce a correct result most of the time. However, this is clearly against the spirit of automated refactoring tools, because they are precisely useful when they can accurately handle difficult situations which can hardly be done by hand.

The following aspects of a language may prevent correct refactoring implementations on it:

- Meta-programming (Macros in C and C++ \cite{33, 34}, evaluation of strings as regular code in JavaScript, evaluation of token lists as regular code in LISP, etc).
- Introspection (also known as reflection). More generally all kind of by-name references in which a name can be built from a string. Note that several uses of introspection (such as serialization) do not depend on specific names are hence refactoring-proof.
- Pointer arithmetic.
- Dynamic typing. A refactoring engine cannot for instance know what types and classes are referenced by other classes.

Note that one of these points, introspection, is a situation that can break the correctness of Java refactoring tools. However, its use is rare, not recommended and easy to locate. It is hence seldom considered as a practical limitation. The situation is very different with some other languages in which the above points are commonly used everywhere around.

In web applications, the guilty language is JavaScript. With meta-programming (possibility to evaluate strings as regular code) and dynamic typing, correct refactoring is most of the time hardly possible to guarantee.

Due to dynamic typing, a simple "rename method" cannot be performed with 100% accuracy as soon as at least another class has a method with the same name. Note that refactoring engines do exist for dynamically typed languages, but in most non-trivial cases the correctness of the result is not guaranteed. Also be aware that some refactoring tools for dynamically typed languages are able to do a “rename method” correctly (with 100% accuracy), however they use a different definition of the refactoring that consists in renaming all methods that have the same name, even if they have different scopes \cite{74}. This definition is unfortunately less useful (consider renaming a method such as \texttt{getName} into \texttt{getUserName}, and imagine all the other unrelated methods named \texttt{getName} that may undesirably be affected in a large project).

**Language Embedding**

The possibility to embed one language within another can become problematic regardless of the programming language. There are two different situations to consider though:

- When the syntax of the embedding language allows one to figure out in a deterministic way when and where code in another language is embedded.

\textsuperscript{6}Furthermore, the trend toward the constant introduction of new languages is continuing as this thesis is being written with the recent introduction of JavaFX.
• When the syntax of the embedding language does not allow it.

The first case is quite common and does not involve any problem, as the embedded language code fragments can be identified automatically:

• JavaScript for example, can only be embedded in HTML at specific locations: within a “\(<script>\)” tag, and within the values of specific attributes such as “onload”, “onfocus”, etc.

• EL expressions can be embedded almost anywhere in JSP (and even in some XML configuration files with JSF!); however they are clearly identified using a specific pattern (“\($\{\ldots\}\)” in JSP, “#\{\ldots\}” in JSF).

• Java code can be embedded anywhere in JSP, but is again clearly identified by the “\(<%\)” and “\(\%>\)” tags.

In all these situations, the embedded language can be identified and recognized as such without ambiguity from the embedding language. Proper refactoring is hence possible (apart from the practical difficulties mentioned in the two previous sections).

The second case on the other hand is problematic. It typically occurs when Java (or JavaScript) is the embedding language. Indeed, embedding HTML, JavaScript or EL expressions within Java code is almost always done using string literals that cannot easily be differentiated from regular strings.

While a possible solution would be to prevent any form of embedding within Java, this would be very restrictive. Indeed, embedding HTML, JavaScript or EL expressions within Java code is usually the only way of producing dynamic content, that is, that depends on the result of non-trivial operations. Obviously the ability of producing dynamic content is usually the main reason for which a web framework is chosen over plain static HTML.

A consequence is that the problem is usually worse than expected because embedding within Java is usually only used in non-trivial cases that cannot be expressed without embedding. Most of the time, the embedded code is constructed using conditionals, loops, method calls and other constructs rather than being just a single string literal constant.

The problems caused by the embedding of a foreign language within Java code are in fact consequences of a more general problem: the overuse of strings in web frameworks, which is more generally due to lack of abstraction.

**Overuse of strings**

This is probably the most important problem with web frameworks. In several, if not all web frameworks, strings are frequently used for things that have higher semantics than just a sequence of characters. The problems begin when strings are used to refer to items that a development tool potentially wants to manipulate automatically (check for syntax errors, perform search for occurrences, and of course, refactor).

It is important at this stage to clearly define the notion of “strings”. Indeed, any language has a textual representation; however a program is never treated as a single big string by a development environment that allows refactoring of the language. To do some refactoring, the program is first converted into a
representation of higher level, such as an Abstract Syntax Tree (AST). Hence we give the following
definition of a “string” (which is independent of the language, as long as the language can be parsed):
Any part of the code that remains a string when the code is converted into an AST (with the “A” for
“Abstract” taken to its full extent).
Hence the following things are considered as strings according to this definition:

• A string literal, such as "Hello world!" in Java or ‘Hello world!’ in JavaScript.

• String literals that occur between tags in HTML, and that are hence displayed by the browser (rather
than being interpreted).

• Textual content (such as labels displayed in the application’s user interface, error messages and
help texts) occurring in XML or properties files.

The following are not considered as strings according to this definition:

• The name of any named item within an XML file. Such a name would be treated like a variable
name in Java: it is converted into a binding during the construction of the AST, at least in an ideal
scenario. A binding is not a string (although it can still encapsulate the original name as a string),
but an object that models or refer to the underlying named construct.

• The name of a tag in HTML or XML. Obviously, like a Java primitive operator, an XML or HTML
tag is modeled as a node of the AST.

• The name of an attribute of an HTML or XML tag. Again, in a suitable abstract representation, an
attribute is modeled as such (like for instance Java modifiers or annotations), and not as a string.

• The value of any attribute of an HTML or XML tag that is not of string data type. In an abstract
representation, the value is modeled by an object of the actual type. Note that it remains possible to
model it as a string, but then there must be a cue on the model to differentiate it from a true string
property (in XML this can be the XML schema for instance).

Another (but less formal) way of understanding this definition of strings is through delimiters. In Java
and JavaScript, all strings are delimited with quotes, and everything else is not a string. In HTML, all the
non-strings are delimited by the “<” and “>” symbols. Hence, everything that is not delimited in HTML
is a string.

While quotes are also present around attributes in HTML, they do not necessarily delimit strings:
the actual data type is defined by the HTML language. For instance, in the HTML fragment “<td
colspan="2", align="center" title="Hello">”, only the value of the “title” attribute
value is a string (and is actually displayed, as a tool tip, by a browser). The “colspan” attribute value
on the other hand is an integer, while the “align” attribute value is of enumerated type. The attribute name
itself is obviously not a string according to our definition, but an identifier.

In XML, the exact data type of any attribute or node content can be defined without ambiguity using
document type definition files (DTD).

Now that we have a clear definition of strings, let us explore how web frameworks are making an
overuse of strings and why this is a problem.

Typed programming languages have always evolved in the direction of abstracting information. Even
in very early programming languages, rather than manipulating raw bits, we manipulate numbers, char-
acters, references or pointers, enumerations, and so on. In web frameworks, even this very first step of
abstraction is missing at various places. The only difference is that strings are now replacing raw bits.

Consider for example some Java code that has to produce some HTML output.
Following are five possible Java codes that all generate the same HTML output (from the least abstract
to the most abstract). Every implementation corresponds to the common practice of at least one existing
web framework:
// Implementation 1 (Java Servlet, Java Server Pages)
out.write("<table><tr><td align="center" colspan="2" nowrap " +
  style="color:#1A1A1A;background-color:green;" +
  "font-family:times,tahoma">" +
  "Test</td></tr></table>);

// Implementation 2 (Java Server Faces)
writer.startTag("table");
writer.startTag("tr");
writer.startTag("td");
writer.writeAttribute("align", "center");
writer.writeAttribute("nowrap", null);
writer.writeAttribute("colspan", "2");
writer.writeAttribute("style", "color:#1A1A1A;" +
  "background-color:green;font-family:times,tahoma");
writer.write("Test");
writer.endTag("td");
writer.endTag("tr");
writer.endTag("table");

// Implementation 3 (DOM-based frameworks)
Element table = new Element("table");
Element tr = new Element("tr");
table.addChild(tr);
Element td = new Element("td");
tr.addChild(td);
td.setAttribute("align", "center");
td.setAttribute("nowrap", null);
td.setAttribute("colspan", "2");
td.setAttribute("style", "color:#1A1A1A;" +
  "background-color:green;font-family:times,tahoma");
td.addChild(new TextElement("Test"));
someExistingElement.add(table);

// Implementation 4 (ASP.NET, Java syntax)
HtmlTable table = new HtmlTable();
HtmlTableRow tr = new HtmlTableRow();
table.rows().add(tr);
HtmlTableCell td = new HtmlTableCell();
tr.cells().add(td);
td.align = "center";
td nowrap = true;
td colspan = 2;
td style.put("color", "#1A1A1A");
td style.put("background-color", "green");
td style.put("font-family", "times,tahoma");
td innerText = "Test";
someExistingHtmlElement.getChildren().add(table);
// Implementation 5 (Java Stones)
HtmlTable table = new HtmlTable();
HtmlTableRow tr = table.addNewRow();
HtmlTableCell td = tr.addNewCell();
td.setColSpan(2);
td.setAlign(Align.CENTER);
td.setNoWrap(true);
        td.style().font().setColor(new Color(0.1, 0.1, 0.1));
        td.style().background().setColor(Color.GREEN);
        td.style().font().setFamilies(FontFamily.TIMES, FontFamily.TAHOMA);
        td.setText(0, "Test");
someExistingHtmlElement.addChild(table);

In the first implementation, everything is just plain string, and there is no abstraction at all. This is
unfortunately still frequently seen in JSP applications, and even in books [46].
In the second example (taken from JSF, but which is also common in ASP.NET), some abstraction is
introduced. However, tag names, attributes and values are still modeled as strings.
The third implementation goes one step further by abstracting at least the tree structure. However it
leaves everything else (tag names, attributes and values) in the form of strings.
Implementation 4 (taken from ASP.NET, but also allowed to some extent by JSF) goes one step further
by abstracting HTML elements into objects, and differentiating different elements using different classes
of a class hierarchy. This allows several attributes to have their type abstracted (the value of “colspan”
becomes an integer and that of “nowrap” becomes a Boolean). However, several attributes and values are
still modeled as strings (namely the value “center” of the “align” attribute and the values of the “style”
attributes components).

Only the last implementation proposes a full abstraction, in which only true textual content is modeled
by strings (note that, although not visible from the above code, constants such as “Align.CENTER” and
“Color.GREEN” are not string constants but enumeration members).

Unfortunately, the last implementation is still far from being the “standard” that is proposed and
used in practice (it is indeed only a small framework “essay” proposed by the author [46]), the common
practice being closer to the second or third implementations [1].

Apart from the last implementation, the various lacks of abstraction introduce several problems, with
respect to automated code analysis and transformations:

• Checking for syntax errors statically (at compile time) is not always possible. For example, only
the last implementation prevents one from setting an invalid value for the “align” attribute of the
table cell.

• If some attribute value, or the "Test" string literal, is computed from some user input, vulner-
ability issues such as cross-site scripting are possible. Only the first implementation is always
concerned, for the other versions it depends on how the underlying framework is implemented;
however the use of strings usually makes it harder to avoid vulnerabilities (see the paper on the
“Stones” framework [46] for more).

• Refactoring of some aspects of the code is difficult, or impossible to perform correctly in an au-
tomated way. For instance, if we want to replace all “times” fonts by “arial” fonts, it is necessary
in all implementations except the last one to look for and replace the occurrences of the “times”
string literal by “arial”. Unfortunately, if the literal “times” occurs somewhere else in the code (for
instance in the return statement of a method), there is no deterministic way of figuring out whether
the string is actually used to denote an HTML font, and is not for example a part of a “regular”
string (such as a message displayed to the user).
Refactoring to HTML 4.0 compliance (or any specific HTML standard) is difficult (it would involve, for instance, replacing the “align” attribute of the table cell by a “text-align” entry in its “style” attribute) for the same reasons as the previous point.

Of course, many of the problems can be mostly solved if the programmer declares string constants for the tag names, attributes and values; and use them everywhere. However a refactoring tool cannot base itself on such an assumption. Again refactoring is precisely useful when the code is not well designed. Furthermore, “extract constant” is typically a refactoring one would like to use to replace all occurrences of the “times” value (or any other tag name, attribute name of value) by a constant.

The problem also concerns other abstractions present in several web frameworks. For example, several web frameworks have the notion of actions to handle button clicks. These actions are typically given through an XML or HTML attribute. If the value occurs directly in the HTML or XML code, there is no problem: the value can be validated against the allowed ones by parsing the XML or HTML, and refactoring is theoretically possible as well. However, web frameworks usually also allow (typically using EL expressions, or similar constructs) providing a value that is dynamically computed by a Java method. Unfortunately, this is usually done by creating a method that returns... a string (the name of the action).

Again in this situation, a simple refactoring that has to replace a given action by another one, or to extract the action name into a constant is not possible to perform with 100% accuracy.

A possible solution would be a framework that maps actions not to Java strings, but to members of a Java enumeration that has to be defined somewhere. To the knowledge of the author, no web framework has yet taken this obvious option. Note that we took actions as an example, but the problem also concerns several other abstractions, when accessed from Java code: JSF’s managed Beans (only easily accessible by their name), outcomes for navigation (in JSF), web page URLs (when redirecting to a page from Java code), table and column names of a database, etc.

Surprisingly, some web frameworks are very close to proposing a “clean” solution in some situations: JSF’s managed beans could theoretically be accessed by their class rather than by their name, because they are always singletons within a given context. In ASP.NET, every web page is associated to a class (named the code behind); hence an URL to a page could be in theory be referenced by the underlying code behind class rather than by a string.

Finally, note that we have taken the example of HTML embedded within Java code, which is an example that is not too problematic. We choose it because it makes it easy to illustrate not only the problem, but also the partial solutions that already exist.

But other situations are much more dramatic, such as the use of JavaScript within Java code: indeed, there is almost no existing and easy to use abstraction, and entire JavaScript fragments are typically produced using plain Java strings. Not only maintenance is difficult, but any kind of JavaScript refactoring is just impossible when embedded into Java strings, especially if the string is built in a non-trivial way. Unfortunately, like for HTML, it is frequently necessary to have non-static JavaScript in a web application. A possible, but extremely heavy solution to overcome this problem is given by the Google Web Toolkit [9], which allows you to write regular Java code instead of JavaScript (with some restrictions though); the framework then manages to automatically translate the Java code into JavaScript.

At the beginning of this section, we highlighted two obstacles to automated refactoring: lack of static typing and overuse of strings. An interesting observation is that the two obstacles are somehow linked: a language such as JSP, HTML or XML that does not allow statically differentiating regular strings from references to named Java methods or properties is in fact not statically typed (at least with respect to the two concerned data types: strings and references to Java item).

---

7They are not true singletons, but within a given context there is only a single accessible instance.

8When we refer to the JSP language, we only refer to the JSP tags within an HTML file, and not to the underlying Java code.
Summary

Today’s web development frameworks introduce several new difficulties to automated refactoring, and some of them cannot be solved without changing the web frameworks themselves.

Among the difficulties that are challenging, but that can be coped with properly, we have:

- The presence of several different languages and syntax. A refactoring tool has to cope with all of them.
- Embedding of one language into another, including, in its simple form, references to items of one language in another. With the condition that the embedding can be identified without ambiguity, for example through specific delimiters.

However, web development frameworks also introduce new aspects that can make even the simplest refactoring impossible to implement correctly:

- Dynamically typed languages, such as JavaScript.
- Embedding of one language into another, that cannot be identified without ambiguity (such as embedding using string literals).
- The use of strings to refer to higher level data types.

Solutions exist to cope with these problems, but are not yet widely spread. Furthermore, they imply changes of the framework itself rather than changes of the refactoring tools:

- Replacement of JavaScript by other languages (such as Java). There are actually two existing solutions:
  - Explicit removal of JavaScript. This is done for instance by the Google Web Toolkit (GWT) that compiles Java into JavaScript, and hence allows one to use Java only.
  - Isolation of JavaScript within “rich components” libraries that aim in providing rich and powerful user interface components without having to write JavaScript explicitly. This solution is unfortunately only relevant as long as you do not need to write your own custom components.

As of writing this thesis, these solutions are not yet mature. This shows up by the fact hand-written JavaScript not only remains necessary in various situations, but is explicitly allowed by such frameworks to work around their limitations.

- Use of delimiters or DTDs to identify language embedding without ambiguity. Examples of delimiters are the “${...}” of JSP and the “#{...}” of JSF, to differentiate EL expressions from regular strings in HTML and XML sources.

- Replacement of strings by appropriate data types, such as enumerations or new classes. An example is the use of object models to generate code in one language from another one (such as the HTML object model of the “Stones” framework presented above). In some simple cases such as programmatic access to managed beans in JSF, class references (such as “getBean(SomeBean.class)”) could be used as well in place of strings (and can even remove the need for type casting using Java 1.5 generics).
7.2 Models

In this thesis we have presented models and algorithms for automated refactoring. In this context, the primary goal of the models (AST, Class Graph, etc) is to analyze the code and to gather information that is required by the actual transformation.

Models also have other purposes that are sometimes surprisingly close. We briefly discuss some of them.

7.2.1 Design and Documentation

The Class Graph is surprisingly close to a standard UML Class diagram. Indeed, UML diagrams are all models of the code. A difference shows up if we consider that there are basically two classes of models:

- Those that fully model the source code, by providing a one-to-one mapping. The typical example is the AST.
- Those that only model a particular aspect of the source code. The Class Graph is such an example.

Automated refactoring cannot be done without at least a full model of the source code. Partial models such as the Class Graph or DFG can be used in the process, but only with the help of a full model such as the AST in order to perform the actual transformation.

UML on the other hand favor partial models that only cover some aspects of the source code. Note however that some UML diagrams can together fully model source code (they are referred to as executable UML), but this is not the general use. Indeed, UML is mostly used for the design and documentation. As such, it has to model the main abstractions of the code, leaving out the implementation details.

An interesting point here is “documentation”. Documentation is about understanding, and is hence close to analyzing, with the major difference that understanding usually refer to the programmers whereas analyzing refers to automated tools.

In both cases, a common point is that a partial model of the code, although it has the drawback of not fully modeling the code, has the advantage of showing meaningful semantics that are not immediately visible from the source code or from a full representation of it through an AST.

Another difference between models such as UML diagrams, that are mainly used for documentation, and models such as the AST and the Class Graph, that are used for analyses and transformations, is that the latter are forced to be close to the underlying programming language in order to be practical. UML on the other hand, usually aims in being independent from the underlying language. Observe that this last statement is somehow nonsensical, especially with respect to executable UML, which is also a language itself; it just happen to have a visual syntax and to be of higher level.

7.2.2 Security

Another use of abstract models is to enforce security. The notion of security covers a large area of different techniques. They range from high level concepts such as authentication and authorization to low level concepts such as the prevention of bugs that can be exploited to gain unauthorized accesses, such as buffer overflows.

Models can be used to enforce the low level security [46]. Not surprisingly, some notions of low level security are close to the notion of correctness in refactoring.

---

The mapping is one-to-one with respect to the semantics. It may not map all syntactic details such as white space.
SQL Injection and SQL Models

SQL injection is a typical case of a low level vulnerability of badly written code. It is a consequence of embedding a language into another, as already discussed in Section 7.1.4. The case we are interested in is when SQL is embedded in Java in the form of strings, a common practice. Consider for instance the following code that would typically be found in an application that allows authentication using a user name and a password:

```java
public boolean authenticate(String userName, String password)
    throws SQLException {
    String sql = "SELECT * FROM User WHERE User.Name = \\
                  ' + userName + ' AND User.Password = \\
                  ' + password + ';
    Statement stmt = connection.createStatement();
    ResultSet result = stmt.executeQuery(sql);
    try {
        return result.first();
    } finally {
        result.close();
    }
}
```

The user name and the password usually come from a text field in which the user can type his user name and password. As long as the user types a “regular” user name and password, such as “toto” and “secret”, the first line of the above code builds the expected SQL statement, which returns a result if and only if a user with the corresponding name and password exists in the database:

```
SELECT * FROM User
WHERE User.Name = 'toto'
AND User.Password = 'secret'
```

Unfortunately, this code has a vulnerability. Assume that a malicious user types “admin” as the user name, and "' || User.Password || '" as the password (including the single quotes, but not the double quotes). The first line of the above code creates the following SQL statement:

```
SELECT * FROM User
WHERE User.Name = 'admin'
AND User.Password = '' || User.Password || ''
```

This SQL statement retrieves the row corresponding to the administrator user without checking the password. Indeed, “User.Password = '' || User.Password || '"" always evaluates to true. This problem is known as SQL injection. It is due to the wrong assumption that the strings `userName` and `password` only show up as strings in the SQL statement. Unfortunately, as the entire SQL statement is itself represented as a string, the `userName` and `password` variables can also contain SQL delimiters (notably the string delimiter, the single quote) and SQL operators that are injected in the statement.

There are many solutions to solve this problem of SQL injection. For other examples and an overview of the possible solutions, refer to previous work on the topic [46].

One solution, which is of interest, is to use abstract object models for SQL statements rather than plain strings. Using a full AST-like model of SQL statements, the above code could instead be written:

```
In SQL, "|" is used for string concatenation.

158
Figure 7.1: AST of the SQL statement "SELECT * FROM Table WHERE Name = @userName AND Password = @password". Items in italic are named parameters used to pass constants.

```java
public boolean authenticate(String userName, String password)
   throws SQLException {
   Query query = new Query();
   query.setTable(UserTable);
   query.setSelection(Query.STAR);
   Criteria criteria = new Criteria(Op.And);
   criteria.add(UserTable.Name, Op.Equals, userName);
   criteria.add(UserTable.Password, Op.Equals, password);
   query.setCriteria(criteria);
   // Execute the query...
}
```

This code actually dynamically builds an AST of the SQL statement, as shown in Figure 7.1. In this example, the AST nodes that correspond to SQL strings are clearly identifiable, and the underlying engine can treat them appropriately.

Such an approach has been taken as an option by the Hibernate framework [4] for instance, as an alternative to the low level SQL and HQL languages. Observe that in this approach, the model of the SQL statement is explicitly expressed in the Java code. Furthermore, this approach is appealing because the programmer no longer has to deal with the SQL syntax directly, and has no risk of mistyping SQL keywords.

The solution in which the SQL AST is “built” directly in the Java code is quite complex, and there are simpler ways of resolving the problem, for instance by using parameters:

```java
public boolean authenticate(String userName, String password)
   throws SQLException {
   Statement stmt = connection.createStatement();
   stmt.setParameter(1, userName);
   stmt.setParameter(2, password);
   ResultSet result = stmt.executeQuery(sql);
}
```
try {
    return result.first();
} finally {
    result.close();
}

This example is comparable to the use of delimiters discussed in Section 7.1.4. Here, a single delimiter, “?”, is used to delimit any SQL literal constant (of string data type in our example). The actual constants however are transmitted in a second step in the form of typed objects rather than in the form of strings. A drawback of this approach is that the table and column names are no longer abstracted and show up as strings.

There is another notable difference between the two solutions. The second one relies on the programmer to actually use parameters, but nothing prevents him from writing the same code without using parameters. Hence the first solution offers stronger security, because the programmer has no choice. In other words, the AST-like solution enforces security at the programming level.

This is comparable to other issues such as pointer arithmetic: as pointer arithmetic is potentially dangerous (and can lead to both bugs and security issues if the bugs can be exploited), it is clear that it should be avoided. The second example would correspond to programming in C without using pointer arithmetic, although the language still allows it. The first example on the other hand is like programming in Java, where there is no risk because pointer arithmetic is not possible at all in that language.

Hence, the use of AST-like structure for building SQL statements can be used to enforce security at the programming level. It is interesting to note that here, the notion of security is close to the notion of correctness: whenever a vulnerability is exploited, the generated SQL statement is in fact incorrect (meaning that it does not perform the intended operation). This is comparable to the notion of correctness of code analysis (such as clone detection discussed in Section 2.2.3): string-based approaches cannot guarantee correctness (100% precision), whereas approaches based on models such as the AST can.

**HTML and XML Models**

It is worth mentioning that what was explained about injection with SQL is in fact also applicable to any other language that is embedded in the form of strings. One example, discussed in Section 7.1.4 is HTML. Because HTML further allows embedding of JavaScript, Java code that produces HTML and includes user input in the process is vulnerable unless it escapes all HTML delimiters from the user input. Indeed, if HTML delimiters are not escaped, a malicious user can try entering some HTML containing “<script>” tags with JavaScript, or even “<object>” tags referring to Java applets or any application present in his own web site. This problem is known as Cross Site Scripting.

As for SQL injection, there are many ways of solving the problem. Again, a way that prevents any cross site scripting at the programming level is to build HTML only by the mean of objects and models. The examples 2 to 5 of Section 7.1.4 are all valid solutions, as long as they do not additionally allow programming like the example 1. In examples 2 to 5, there is sufficient modeling to distinguish raw text from HTML building blocks, and raw text can hence be escaped properly.

**7.3 Summary**

This chapter explored several related areas. In a first part, we showed how techniques similar to those discussed in this thesis are used in various other areas such as compilation, optimization, profiling, debugging, etc. We also highlighted future challenges introduced by new trends in web applications.
In a second part we showed that analyses are linked with understanding, and correctness with security, and that both aspects can be managed in a similar way: using abstract models.
Chapter 8

Conclusion

8.1 Summary

The notion of a “finished product” is rare because existing software constantly evolve. In practice, new features, modifications and adaptations are permanently requested. A consequence is that no initial design, however good, can accommodate all the possible future changes in a real-world project. The agile methodology takes this fact as granted and proposes tools that aim in coping with change rather than defending against it.

One category of these tools is refactoring, or changing an existing design. Refactoring has lead to refactoring tools, which helps in adapting the existing code automatically in order to be kept synchronized with a change of the design.

Refactoring tools, like any software, also evolve over the years. Hence they need to be refactored themselves. This thesis discussed an evolution of refactoring tools, namely the evolution toward more complex transformations.

The need for an evolution was motivated by a complex refactoring: forming a template method. Exploring this refactoring showed that the existing models and techniques were not suitable to solve the problem. New models and algorithms had to be introduced, such as the code differentiation process. Hence it was necessary to refactor existing algorithms toward more complex ones. Other processes on the other hand had to be refactored toward simpler versions that are more suitable to the refactoring process, such as the data and control flow analyses used for method extraction.

In Chapter 1, we motivated the need for refactoring and explained the broader context in which it is used. Refactoring is not a new process, and the state of the art regarding automated implementations was presented in Chapter 2. Chapter 3 presented a concrete case study: a difficult refactoring, forming a template method. This case study showed the limitation of existing approaches and sought for new ones.

Extensions of existing approaches as well as new approaches have been presented in three chapters, corresponding to three steps of the implementation of our case study. Chapter 4 discussed the problem of differentiating two method bodies, as an extension of the problem of clone detection. Chapter 5 discussed the problem of data and control flow analysis. Here we proposed a “light” alternative to the CFG and PDG-based analyses. Chapter 6 finally coped with a new issue, namely on how to automatically solve minor problems rather than systematically reporting them as errors to the user.

Analyses, models and transformations have been used to implement our case study. However they also have other applications in many other areas. Conversely, other areas, such as web programming, still seek for additional research. This was addressed in Chapter 7.

We continue by summarizing our contributions, making a critical analysis of our work, and highlighting future work.
8.2 Contributions

Refactoring tools existed long before writing this thesis, and as such most of our work consist in extensions and improvements over existing techniques. In this section, we first summarize all our contributions. In a second part, we discuss the different parts of the implementation, namely what has been reused, was has been implemented, and what remains to be done.

8.2.1 Improvements and New Techniques

Here is a summary of our main contributions:

- **Code analyses**
  - A new clone detection algorithm, based on a post-order traversal of the AST and the LZ77 algorithm (Section 4.1);
  - An adaptation of the clone detection algorithm to perform code differentiations, using the LCS algorithm (Section 4.2);
  - An algorithm for guessing semantically equivalent variables that have different names in two code fragments that are only partially matching, using bipartite graphs (Section 4.3);
  - A new and fast algorithm to perform the control and data flow analyses required for method extraction, using Boolean flags and expressions (Chapter 5);

- **Code transformations**
  - An approach in which preconditions that are not satisfied are resolved automatically whenever possible rather than being reported as errors to the user. An overview of existing solutions was presented (Section 6.3);
  - A formal illustration of an existing way of resolving multiple exit paths, using the FOOD control and data flow model (Section 6.2.3);
  - A new way of resolving multiple results, by splitting the methods to extract into multiple ones (Section 6.3.2).

All these contributions, combined with reuse of existing techniques and implementations, resulted in the realization of a working implementation of the “form template method” refactoring.

8.2.2 Implementation

The implementation of the “form template method” refactoring has been realized as a part of this thesis, as an Eclipse plugin. Although the implementation is not a full-featured plugin with a user interface (actually it reads hard-coded files and prints the results on the console), it provides a proof of concept for most of the presented new algorithms.

The implementation is based on the following existing technologies:

- Java standard edition, version 1.5;
- Eclipse plugin architecture (based on Eclipse 3.4);
- Eclipse JDT (Java Development Tools), from which the Java parser, the Java AST and Class Graph implementations, and the AST rewriting tools were reused.

The following techniques presented in this thesis have been implemented:
• The full clone detection algorithm based on the LZ77 algorithm, as discussed in Section 4.1.
• The full code differentiation algorithm, as discussed in Section 4.2.
• The full data and control flow analysis using flags and Boolean expressions (as presented in Chapter 5), including the handling of conditionals, loops, and jump statements.
• The new proposed way of resolving multiple results, presented in Section 6.3.2.
• Prototype “extract method” and “form template method” implementations based on the above.

A few things on the other hand have not yet been implemented, and were simulated by hand, or processed by other tools, to perform the tests and validations:

• The matching of renamed variables presented in Section 4.3. The algorithm alone was actually implemented and tested, but has not been integrated in the refactoring algorithm.
• The existing technique of resolving multiple exit paths (Section 6.2);
• The existing techniques of resolving multiple results; only the new proposed one was implemented.
• In the whole algorithm, specific Java constructs are not, or not fully handled, such as the switch statements, exceptions, synchronized blocks, generics (Java 1.5) and inner classes.

Apart from the implemented features of the algorithm itself, the following is future work to be done:

• A complete user interface and a better integration within the Eclipse development environment;
• Systematic tests on more and larger inputs. Only small examples have been tested, including all the examples presented in this thesis.

8.3 Analysis

This thesis was mainly driven by a concrete problem, namely the correct implementation of a concrete and difficult refactoring. As this thesis evolved, this approach has shown to have both advantages and drawbacks. Among the drawbacks, we observed the followings:

• The chosen algorithms and models were targeted and focused to the concrete problem, and it is not yet clear how widely they can be generalized to other similar problems. One may want to see a more general, language independent “framework” for refactoring, or a more general “picture” of new and future refactoring problems, algorithms and models.
• We chose a pragmatic approach that allowed us to test our ideas. However, it is difficult to tell anything about the properties of the proposed algorithms. This is still future work to be done, for instance using formal approaches rather than pragmatic ones.
• We sometime proposed new alternatives to solve problems for which other solutions already exists (flow analysis using flags in Chapter 5 when DFG and CFG already do it; a new way of resolving multiple results in Section 6.3.2 when many other solutions exists). This is debatable. On one hand, software engineering practices almost always favor reuse; on the other hand, Agile programming methodologies favor refactoring any existing solution that is suboptimal, in order to prevent both under and over engineering.
• Only the evolution toward more complex refactorings was considered. More work needs to be done regarding other evolutions, such as toward more complex languages and frameworks (briefly highlighted in Section 7.1.4), and eventually toward more complex paradigms such as distributed programming.

However, the chosen approach has also shown to have many advantages:

• We ended with a working algorithm that solves a concrete problem.

• As a concrete problem was targeted rather than an approach, algorithm or model; we were forced to find new approaches, algorithms or models whenever the existing ones showed to be inadequate or insufficient to solve the problem. This prevented us from falling in the trap of an elegant and beautiful theory that is useless or too limited in practice.

• The problem to solve was clearly defined. Hence we could cover the whole process in both high level concepts and low level implementation details.

• The ambition of having a working implementation forced us to choose solutions that are realizable in a reasonable amount of time and resulted in an implementation that runs with reasonable speed and memory footprints. One example was the development of the flags-based data and control flow analysis against the tedious creation of a full Control Flow Graph or Program Dependence Graph of the code. Another one was the reuse of existing stuff such as the fast parser and the Abstract Syntax Tree model of Java code provided by the Eclipse development environment.

• Most of the algorithms and models were implemented and hence could be tested on concrete code samples.

Finally, we can gather a few observations and “lesson learned”:

• The pragmatic approach, driven by a concrete goal, showed to be fruitful: parts of the algorithm that initially looked like details at a first glance, turned out to be complex enough to result in the elaboration of new algorithms or models. Examples are the matching of renamed variables, or the resolution of multiple results.

• The data and control flow analyses have shown to be the most complex part of the algorithm. Exploration of existing literature showed that this problem is frequently overlooked or just ignored: most of the time the CFG of PDG is just assumed to be available, and its construction is hence not discussed.

• While refactoring looks like a specific domain that requires a domain specific language at a first glance, our case study revealed the necessity of using various general-purpose algorithms, that are not specifically related to refactoring or code transformation: LZ77, Longest Common Subsequence (LCS), maximum weighted matching of bipartite graphs, Lowest Common Ancestor (LCA) in a tree, etc.

• As more complex refactoring are explored, it looks like the number of ways of executing the transformation correctly increases as well. An example is the different ways of resolving multiple results in the “extract method” refactoring. Another one is the degree of optimization in some analyses, such as the recognition of renamed variables, or the use of LCS versus an LZ77-based heuristic. Simpler refactorings, such as “rename method”, do not exhibit such degrees of freedom.
• Thank to the restrictions of the Java language over other languages (static typing, absence of pointer arithmetics), static analysis and refactoring transformations can be performed with near 100% accuracy\(^1\). However, new frameworks and languages (especially for web development) look like a big step backward in this respect, as shown in Section 7.1.4.

• Abstraction and models (AST, Class Graphs) are mandatory for correct implementation of code transformations.

To conclude, the proposed case study was solved and resulted in several new challenges and corresponding solutions, in the form of novel models and algorithms. However, as in any new technology, it has not yet a degree of maturity with which we can easily speak about its properties (effectiveness in the long term, scalability, flexibility) and compare it with related approaches.

This naturally brings us to the last section.

### 8.4 Future Work

We have explored a complex refactoring as a case study: forming a template method. This refactoring is further decomposed into other, smaller refactorings, namely “extract method”, “pull up method”, and eventually “rename variable”. There are of course many other refactorings to explore that have not yet been implemented, and this could be a future research direction. In particular, our case study has shown that existing techniques are not sufficient to implement the form template method refactoring, and required extensions. It is however too early to say whether and to what extent the new introduced approaches can or cannot be reused for other, even more complex refactorings.

While this was not in the scope of this thesis, research exists regarding refactoring composition. More precisely, this research is concerned with the global pre- and postconditions of two or more refactorings applied one after the other. It makes sense to apply and extend these researches to the problem of forming a template method, because this refactoring is precisely a composition of multiple refactorings (namely method extractions, and pull up method).

As our work was targeted on the algorithm itself, user interface issues have not been addressed. However we believe it to be a large part that remains to be done in order to get a full-featured implementation. The design of the user interface raises several issues, such as:

• To what degree can the user control the operation. It seems obvious for instance that the user has to choose the name of the template method. However, other decisions can make the tool quite complex if they are based on user interaction, such as:
  
  – Fixing or adjusting the differentiation process;
  – Fixing or adjusting the matching of renamed variables; and choosing the new unique name for each variable pair;
  – Choosing the strategy to use in the presence of multiple results, among the possibilities discussed in Section 6.3;
  – Choosing the name of each delegate method.

These interactions all correspond to decisions for which an optimal choice cannot be made programmatically (although a good guess is possible), as it depends on the semantics of the code.

• How to allow the selection of two code fragments;

• How to report the errors in the most useful way in case the refactoring fails;

\(^1\)The reflection API is one of the rare remaining problems of Java with respect to 100% accurate refactoring.
• How to display and navigate among the results of multiple alternatives, if the user is allowed to choose one among them;

By exploring a complex refactoring that has not yet been implemented, we explored the evolution of refactoring tools toward more complex refactorings. There are two other evolutions that could also be addressed:

• The evolution toward new languages, paradigms and frameworks. A brief overview was given in Section 7.1.4;

• The evolution toward generalization, or language-independent algorithms. The use of models such as the AST and Class Graph already helps. However, as shown in Section 7.1.4, new frameworks are involving languages that are more and more different: statically typed (Java) versus dynamically typed (JavaScript), imperative versus declarative (XML, HTML), etc.

One of the key models used by our algorithm was the AST. The AST is also useful for many other tasks, such as compilation, searches for references, code metrics, etc. It would be interesting to study on whether or not the new algorithms and models we have introduced also have other uses than in the “form template method” refactoring.

Finally, we took a pragmatic approach to the problem, which allowed us to get a working solution. On the other hand, a more formal approach would be necessary to discuss our algorithm in terms of its properties (preconditions and postconditions) and correctness. Formal approaches may eventually find out that parts of our algorithms are wrong or suboptimal, or may just need adjustments and extensions to cope with future programming languages. Following the Agile development philosophy, this would not be a problem: as with any real-world application, in such a case this thesis and the underlying research would just need to be refactored themselves.
# Appendix: Index of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D</td>
<td>Three Dimensional</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASP</td>
<td>Active Server Pages</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DEMUX</td>
<td>Demultiplexer</td>
</tr>
<tr>
<td>DFG</td>
<td>Data Flow Graph</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Bean</td>
</tr>
<tr>
<td>EL</td>
<td>Expression Language</td>
</tr>
<tr>
<td>FBY</td>
<td>First, followed by</td>
</tr>
<tr>
<td>FOOD</td>
<td>First-class Object Oriented Data flow</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>GWT</td>
<td>Google Web Toolkit</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HQL</td>
<td>Hibernate Query Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>J2EE</td>
<td>Java Platform, Enterprise Edition</td>
</tr>
<tr>
<td>JDT</td>
<td>Java Development Tools</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Faces</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td>LCA</td>
<td>Lowest Common Ancestor</td>
</tr>
<tr>
<td>LCS</td>
<td>Longest Common Subsequence</td>
</tr>
<tr>
<td>LISP</td>
<td>List Processing Language</td>
</tr>
<tr>
<td>LZ77</td>
<td>Abraham Lempel, Jacob Ziv, 1977 (algorithm by)</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplexer</td>
</tr>
<tr>
<td>NAC</td>
<td>Negative Application Condition</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>PDG</td>
<td>Program Dependences Graph</td>
</tr>
<tr>
<td>QL</td>
<td>Query Language</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XP</td>
<td>Extreme Programming</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformations</td>
</tr>
</tbody>
</table>
Bibliography


[33] Alejandra Garrido: *Program Refactoring in the Presence of Preprocessor Directives*, PhD at the University of Illinois at Urbana-Champaign, 2005

2 All web references were last visited and checked on April, 2009.


[49] Günter Kniesel: *ConTraCT - A Refactoring Editor based on Composable Conditional Program Transformations*, Pre-proceedings of GTTSE, 2005


171


[67] Matthias Rieger: *Effective Clone Detection without Language Barrier*, PhD at the University of Bern, Switzerland, 2005


172


Curriculum Vitae

Contact
Nicolas Juillerat
Rue de la Carrière 8
CH - 1700 Fribourg

Phone: 026 300 9285
E-mail: nicolas.juillerat@unifr.ch, moi_question@pitchtech.ch

Birth date: 15.04.1978

Education

• Primary and secondary school in Marly (9 years)
• Ste-Croix Gymnasium in Fribourg (4 years), type C Matura (sciences)
• University of Fribourg (4 years), main study: Computer Sciences, secondary study: Mathematics.
  – Bachelor work in computer sciences: Application for the semi-automated planning of courses and exams.
  – Seminar work in mathematics: Data compression using arithmetic coding.
• Diploma in computer sciences (June 2001)
  – Master work: A real-time sound effect server (SoundEngine).
  – Skills: C++, Digital Signal Processing, Real-time, FFT, Binary file formats (AIFF, WAVE, GIF, JPEG, SUN Audio, Midi), Digital video and sound, Remote Procedure Call (RPC), GNU/Linux, NetBSD, Solaris, Irix, QNX, Cygwin, PowerPoint, LaTeX.
• PhD at the University of Fribourg, supervised by Prof. Béat Hirsbrunner.
  – PhD title: Models and Algorithms for Refactoring Statements
  – Skills: Distributed systems, Multi-agent systems, Pervasive computing, Coordination models, Artificial intelligence, Graph theory, Graph transformations, Programming languages, Class Graph, Abstract Syntax Tree, Data and control flow analysis (DFG, CFG, PDG), Refactoring, Clone detection/removal, Eclipse JDT, Eclipse Plugins.
Employers

- **University of Fribourg** (1.6.1999\(^3\) to 31.8.2008, several casual employments), computer sciences department, for application development and web site maintenance.

  **Skills** Java, RMI, CodeWarrior, FileMaker, Servlets, HTML, Apache

- **Compagnie Financière Michelin** (1.1.2000 to 30.7.2000), 20% as Visual Basic 6 / Access developer

  **Skills** Visual Basic, Access, Word, Excel, SQL

- **RAAF-CQS SA**: 50% as software developer (15.8.2001 to 31.10.2003)

  **Skills** Java, Web services, XML, ODBC, JBuilder, Eclipse, SourceSafe, Enhydra, Dataflow

- **Caritor Suisse SA**: 60% as software developer (1.1.2004 to 31.10.2006)

  **Skills** ASP.NET, Visual Basic.NET, JavaScript, Visual Studio.NET, SQL Server, SQL

- **ETH Zürich**: 60% as scientific assistant for Audio/OpenGL software development. (1.10.2006 to 31.4.2008)

  **Skills** C++, Java, Open Sound Control, MIDI, ALSA, OpenGL, Digital Audio Effects, XML DOM, Real-time, Low latency programming

- **Tecost SA** (current position): 60% as software developer (15.9.2008 - )

  **Skills** JSP, Struts, Spring, Hibernate, Postgres, UML (MagicDraw), Maven 2, JavaServer Faces, JPA, Facelets, Ajax4jsf, RichFaces

**Main Skills**

**Spoken languages** French, English, little German

**Programming languages** Java, C++, Modula-2, Visual Basic, Visual Basic .NET, Delphi, Pascal, C, C#, Scheme, SQL, OPL, Basic, ASP, JScript, Assembly 68000

**Frameworks** JSP, JPA, JavaServer Faces, Facelets, .NET 1.3-2.0, Eclipse, Struts, Spring


**File Formats** (internal structures) HTML, XML, LaTeX, GIF, ZIP, RIFF-WAVE, IFF-AIFF, Sun Audio, JPEG, MIDI, etc

**Operating systems/APIs** JVM, .NET, Windows 95/98/NT/XP, MacOS (7-9/X), GNU/Linux, Solaris, NetBSD, .NET, X-Windows, EPOC, OS/2, AmigaOS, Atari (TOS)

**Specializations**: Audio and image processing, computer graphics, refactoring and code manipulations, binary file formats, framework design/development, data compression, real-time systems.

**Hobbies**: Programming, acoustics, DSP, digital video, graph theory, Sudoku, real-time, audio effects, artificial intelligence, programming frameworks, chess and coffee drinking :-)

\(^3\)All dates are in DD.MM.YYYY format.
Publications

PhD related


• Nicolas Juillerat, Béat Hirsbrunner: *FOOD: An Agent-Oriented Dataflow Model*, 7th International Conference on Artificial Intelligence and Soft Computing, pp. 835–840, 2004

Other


