Toward an Implementation of the "Form Template Method" Refactoring

Nicolas Juillerat
University of Fribourg
nicolas.juillerat@unifr.ch

Béat Hirsbrunner
University of Fribourg
beat.hirsbrunner@unifr.ch

Abstract

This paper presents an implementation of the "form template method" refactoring. This transformation has not been automated yet, but has many similarities with other transformations such as clone detection and removal or method extraction. Forming a template method is a difficult process because it has to deal with code statements directly. Few abstractions and algorithms have been investigated yet, compared to transformations dealing with higher level aspects such as the classes, methods, fields and their relations. We present a new algorithm that performs this transformation in a semi-automated way on Java programs. We state the difficulties inherent to this transformation and propose solutions to handle them.

1. Introduction

It has been widely accepted in the software engineering community that any software is subject to entropy: the design of a system is constantly changing while it is being developed, meaning that the initial architecture tends to gradually degrade over time.

Refactorings [10, 14] are small semantics preserving code transformations. Their aim is to counter care this problem, by helping the developer to cope with changes of a design over time. It is now becoming standard for a development environment to provide at least a few refactoring implementations, such as renaming fields or methods, introducing delegates, etc.

In this paper, we present a new algorithm that performs the "form template method" refactoring. This is a transformation that takes as input two methods that are similar, but not exactly the same, such as the following:

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

```
public void skewAt(Point center,
    double amount) {
  translate(mult(center, -1 - 4));
  amount = skew(amount);
  translate(center);
  normalize(amount);
}
```

Listing 1

The purpose of the refactoring is to build a *template method* that captures all the statements that are the *same* in both methods. The two methods typically belong to two classes extending the same parent class. The differences are extracted into new methods of both classes and the template is then pulled up into the parent class. It may look as follows in our example:

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

The methods **d1** and **d2** contain the differences. They are abstract in the parent class. They are implemented in both subclasses and are invoked in a polymorphic way from the template method in the parent class. This example will be used throughout this paper as an illustration of our algorithm.

Most other existing refactorings are dealing with classes, fields, methods and their relations [10]. They are thus limited to the "declarative" part of a program. Excellent models and languages have been developed to help their implementations [3, 5, 6, 7].

Forming a template method on the other hand is a transformation that has to deal directly with code statements, or the "executable" part of a program. Models and languages are much more limited in this area. The commonly used representation of code statements is the Abstract Syntax Tree (AST) [3]. This representation is quite poor in expressing information that is relevant for the transformation of statements. As a result, few refactorings dealing with statements have been successfully implemented yet.

The process of forming a template method is closely related to the process of detecting and removing clones. Both problems can be solved by similar algorithms, but only clone detection and removal has been investigated yet [2, 8, 13]. Clone removal is also closely related to the "extract method" refactoring [12, 16].

In this paper we present an algorithm to form a template method. The algorithm is based on existing techniques used for clone detection and removal. Our contributions are hence the following:

- The process of forming a template method has some notable differences compared to the process of clone detection and removal. We state them and propose new or modified algorithms to handle them.
- We propose a novel algorithm structure based on three steps instead of the usual two steps (detection and extraction). This structure gives us additional freedom that can be exploited to improve the overall quality of the transformation.

The rest of this paper is structured as follows: in section 2, we give an overview of the main steps of our algorithm. In sections 3 to 5, we explain the implementations of the individual steps in details. In section 6, we present the current state of a concrete implementation of our algorithm as an Eclipse plugin, and we give preliminary results as well as future working directions. We then compare our work with related previous research in section 7 and conclude in section 8.

2. Overview

This section presents the overall structure of our algorithm and the motivations behind it. The detailed implementation is then explained in sections 3 to 5.

2.1. Structure of the Algorithm

Our algorithm is basically structured in three steps. Note that this subdivision is not limited to the process of forming a template method, but can also be applied to the process of detecting and removing clones, which is very similar. The steps are the following:

- Detection of similarities and differences
- Resolution of constraints
- Extraction of methods

The first step is obvious: a template method is a method that captures every common statement between two different methods. In order to form it, we have to identify these common statements. A similar analysis is necessary for clone detection, except that we are not working with a pair of methods but with an entire program.

Various algorithms have been investigated for this analysis in the field of clone detection [4, 13, 16]. Our solution is mainly based on previous work and is discussed in section 3. It competes with the best existing approaches in term of efficiency and speed, at the expense of some additional complexity in the algorithm.

The last step of the process (we leave the second step for the end of this section) is to perform the methods extractions. It consists in extracting subsets of consecutive statements into new methods. When detecting and removing clones, subsets of *duplicated* statements are extracted. When forming a template method, subsets of *different* statements are extracted. In both cases though, the process is similar, and is not different than applying the "extract method" refactoring multiple times. This transformation is already deeply covered in the literature, and our implementation is based on existing research.

The second step of the process finally, the resolution of the constraints, is the main novelty of this paper.

If we recall the initial problem we have to solve, it seems at a first glance that only the first and third steps are necessary. After all, we have to identify differing statements between two methods in a first step, and then we have to extract them into new methods. We therefore have a process that looks like the composition of two steps: an analysis followed by a transformation. Indeed, the problem of detecting and removing clone, as its name suggests it, is usually presented that way.

We now give motivations for the introduction of the additional intermediate step and explain its purpose.

2.2. Motivations

The first reason why we choose to introduce an additional intermediate step is the following: the process of extracting a method (used in the last step) is simply not possible with arbitrary subsets of statements. More precisely, the "extract method" refactoring has various preconditions [2, 4], and can only be performed if all of them are fulfilled. Furthermore, previous research in the field of clone detection shows that automatically detected differences or duplications break at least one of these preconditions on nearly half of the cases [13].

Concretely, the purpose of the intermediate step we are introducing is to solve this problem by checking for ranges of statements that cannot be extracted, and to modify them in such a way the extraction becomes possible. Not surprisingly, the actual modifications are driven by the various preconditions of the "extract method" refactoring.

Existing tools for clone detection and removal have used various alternate approaches to solve this problem. The simplest one is to report the broken preconditions to the user, who can then resolve them before the extraction [3]. Other authors have investigated various tricks to ex-

tract "difficult" methods on the C language, to increase the chances that the method can be extracted successfully [12]. Finally, it is also possible to modify the first step in such a way it only produces results which are suitable for the extraction [16].

The introduction of a second step between the detection and the extraction gives us more flexibility: in the last step, nothing forces us to extract *exactly* the fragments that are detected in the first step, as long as the *final result* remains correct. More precisely, the only hard constraint on the final result is that only common statements can be left in the template method. Else it cannot be pulled up safely in the parent class.

Conversely, only different statements *should* be extracted but this is *not* a hard constraint. It does not prevent the template method from being created and pulled up.

A second motivation for the introduction of the second step is that we can usually transform the statements in several different ways to allow the extractions. As a result, a full-featured and interactive tool can present *multiple alternatives* to the user, leaving him the choice of the one to apply.

Section 3 briefly describes the first step of our algorithm, the detection of differences and duplicated statements. Section 4 presents the second step, which consists in modifying the detected fragments so that they can be extracted safely. Section 5 deals with the last step, the extraction of the methods.

3. Detection of Differences

In this section, we investigate the first step of our algorithm in details. This is a purely analytical step, whose purpose is to identify the fragments of code that are duplicated and those that are different between two methods. By a *fragment* of code, we mean an expression or a list of *consecutive* statements.

We only focus on the detection of *duplicated* fragments: the fragments that are *different* are then just the remaining ones.

Our choice is to use a list-based approach that has been successfully used for clone detection [4] with a few adaptations. This approach can be summarized as follow:

- Parse the code into an Abstract Syntax Tree (AST)
- Use a post-order traversal to get it as a token list
- Apply a fast detection algorithm on the resulting list

The last step for clone detection is to use the LZ77 algorithm [17] to detect duplicated code statements[1]. In our case, we cannot use the same algorithm for the last step. First because we have *two* different lists corresponding to the bodies of the two methods, second because we can only consider duplicated fragments that occur *in the same order* in both methods.

Our proposal is to use a *differentiation* algorithm instead of the LZ77 algorithm, such as the one used in the Linux **diff** command [18].

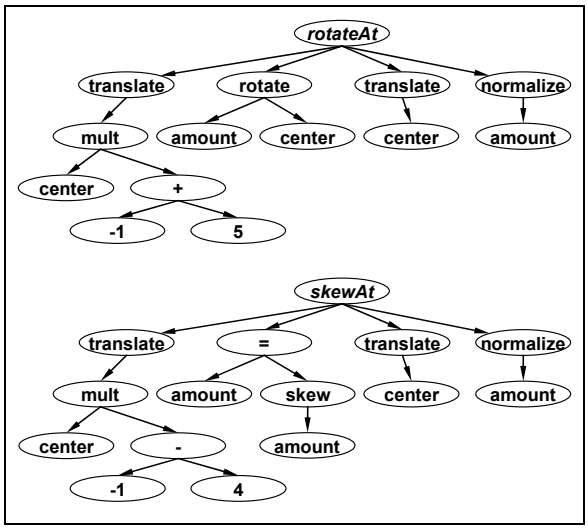

Figure 1 : ASTs of the two initial methods

Let us illustrate the process. Figure 1 shows the ASTs of the two methods of Listing 1 presented in the introduction. After the post-order traversal, we get the two following lists of tokens:

```
[center, -1, 5, +, mult, translate,
 amount, center, rotate, center,
 translate, amount, normalize]

[center, -1, 4, -, mult, translate,
 amount, amount, skew, =, center,
 translate, amount, normalize]
```

A differentiation algorithm immediately reveals that the three following non-trivial sublists occur in both token lists:

```
[center, -1]
[mult, translate, amount]
[center, translate, amount, normalize]
```

These sublists correspond to statements that are *the same* in both methods. The statements that are *different* are formed by the remaining tokens. These statements are those that we will have to extract in new methods. In our example, they correspond to the following sublists:

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

These sublists do not necessarily correspond to subsets of statements that can be extracted safely. In this example for instance, they do not even correspond to single ex-

[1] The LZ77 algorithm is mainly used in the field of data compression. It is part of the implementation of various popular compression techniques such as **gzip**.

pressions. We will deal with this problem in the next step of the algorithm, described in the next section.

There is an important issue to remember at this stage: when we are using a post-order traversal to get a list of tokens, we are not *converting* the AST into a list; we are rather creating a *view* of the AST as a list. In other words, each token of the list is still a node of the AST as well: it not only knows its position within the list, but also its parent and child nodes within the AST. This is an important fact because we will need the original tree structure in the next steps of the algorithm. The list representation is only constructed in order to apply a fast differentiation algorithm, but we do not loose any structural information from the AST in this process.

A differentiation algorithm has an $O(n^2)$ *worst-case* complexity. But good implementations typically have a nearly linear complexity on average [18]. AST-based approaches have been investigated for clone detection and typically give an $O(n^2)$ *average* complexity [16]. Thus, our list based approach is expected to give slightly faster results than an approach directly based on the AST.

4. Application of Constraints

In this section, we describe the main part of our algorithm, the second step. Its purpose is to transform the similarities and differences detected in the previous step, so that all the differences can be safely extracted into new methods in the next step.

The implementation consists in applying a list of *constraints* that are mostly independent from each other. The purpose of the first constraint we present is to resolve a side effect of using a list-view of the AST for the differentiation. The purpose of the other constraints is to resolve preconditions of the "extract method" refactoring.

4.1. Completing Expressions

In the first step that was described in section 3, we used a token list to look for differences and similarities between two methods. Because a list has much less structural information than an AST, we ended up in our example with sublists of statements that do not correspond to single expressions or to sequences of consecutive full statements:

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

The purpose of the first constraint is to extend these sublists so that each of them corresponds either to a full expression, or to a list of consecutive full statements. Indeed, these are the only kinds of fragments we can safely extract into new methods.

The implementation requires the initial AST structure and basically works as follows for a given sublist:

- Search the first common ancestor (in the tree structure) of all nodes of the sublist.
- Extend the sublist with all missing descendents of the common ancestor.

The effective result after this process is that a sublist always contains all the nodes of a particular *complete subtree* of the AST. The common ancestor is the root of that subtree.

Any complete subtree, by the definition of the AST, corresponds to a full expression. The extraction is therefore possible after this process (apart from any other broken preconditions of method extraction).

Because the token list was generated using a post-order traversal, all the nodes of a given complete subtree are consecutive in the token list and effectively correspond to a sublist (and not to an arbitrary subset).

There is only one special case to deal with: when the common ancestor is a block node. A *block* node corresponds to a list of statements within braces. Such a node is used for instance to model the body of a loop. If the common ancestor is a block node, it is *not* necessary to include all its descendents. Only the descendents of its *children* need to be included, for which at least one descendent already belongs to the sublist. A block is indeed the only node that does not model an expression but a sequence of statements.

By applying this constraint on our example, we get the following new sublists:

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

Observe that every sublist now corresponds to a complete subtree of the ASTs illustrated in Figure 1. These sublists of statements can hence be extracted safely. The process of extracting these expressions will be discussed further in section 5.

This example is a good illustration of the freedom that we exploited: because we are not forced to extract exactly the detected fragments, we chose to extend them to make the extraction possible. The drawback is that we capture slightly less common expressions in the template methods: both the "`-1`" and "`amount`" expressions are still duplicated in the extracted methods. They were added while completing the expressions to extract.

These two expressions are very small in this particular example. In practice we may have much bigger expressions that are duplicated. But the advantage is clearly worth the drawback: without the step detailed in this section, we could not even extract any method and the whole transformation would be impossible.

4.2. Multiple Outgoing Data Flows

We now discuss another transformation of the detected sublists of different statements. Its purpose is to resolve a precondition of method extraction.

When we extract consecutive statements or an expression into a new method, we have to pass all local variables that are read as arguments to the method, and to return all local variables that are written and read afterwards as results of the method. Unfortunately, the Java language does not allow a method to return more than one value.

There are various ways to overcome this problem. One possibility could be to return an array, or a class containing all the results [15]. Another solution could be to pass the values by reference; this is not directly possible in Java, but this can be simulated easily by encapsulating the values into "container" objects. A more "brute force" solution is to convert local variables to instance variables so that they are visible by both methods. This solution produces methods that are neither re-entrant nor thread-safe, but can be relevant in some cases, especially when combined with the "create method object" refactoring [9]. A last possibility could be to enlarge the fragment to extract until it eventually corresponds to statements that are writing no more than one variable [4].

Our algorithm implements several of these approaches and presents the corresponding results to the user, who can select the best choice. Additionally we implemented an additional and novel approach that we want to describe here. It is based on previous work on clone removal [4], but has been adapted to fit the problem of forming a template method.

Consider the following statements, and assume that the first five lines (all lines except the last one) have been detected as a fragment to extract into a new method by the process discussed in section 3:

```
min = x - y / 2; // extract from here …
y = y * 2;
max = x + y / 2;
middle = (min + max) / 2;
max = max + 1 ;  // … to here
doStuff(min, middle, max, x, y);
```

Listing 2

Four variables are written in the fragment to extract and are read afterwards: **min**, **y**, **max** and **middle**. As previously suggested, extending the fragment with the last line could solve the problem if the variables are never read again afterwards.

The alternate approach we propose is to split the fragment into *multiple* smaller fragments, so that each of them writes only one variable. We can then safely extract each fragment separately. When applied on Listing 2, this yields to the following result:

```
min = getMin(x, y);
y = scale(y);
max = getMax1(x, y);
middle = getMiddle(min, max);
max = getMax2(max);
doStuff(min, middle, max, x, y);
```

Listing 3

This scheme has been shown to work well for clone removal [4]. But things are surprisingly much more complicated when forming a template method. The reason is that in clone removal, we are interested in extracting duplicated code statements, that is, statements that are the *same*. When we form a template method, we are interested in extracting pairs of consecutive code statements that are *different*.

Recall that we are starting from a pair of methods. The initial code snippet shown in Listing 2 belongs to *one* of the two methods. But we also have to deal with the corresponding snippet in the *other* method at the same time. As a consequence of the process detailed in section 3, its fragment to extract is necessarily different. Imagine that it looks as follows:

```
min = x + 2; // extract from here …
middle = x * y;
max = min + middle;
min = min + 1;
y = x + min; // … to here
doStuff(min, middle, max, x, y);
```

Listing 4

We are in trouble, because the two snippets (Listing 2 and Listing 4) are not writing the same variables in the same order. Using the same scheme as for clone detection on each snippet, we may get different extracted methods with *different arguments and results*. Because the template method has to call these methods in a polymorphic way after the whole transformation, we *cannot* allow them to have different signatures.

We now give an extension of the discussed scheme that works even for two completely different code fragments. The main idea is to identify all the write accesses to variables in both code snippets. We model them using two *write-accesses* lists, containing the variables that are the targets of the assignments. With our previous example we get the two following *write-accesses* lists:

```
[min, y, max, middle, max]
[min, middle, max, min, y]
```

The first list corresponds to the write accesses in Listing 2 and the second list to the write accesses in Listing 4.

We can then proceed with one variable after the other. Intuitively, because the two snippets are first writing the **min** variable, a pair of methods can be extracted for it without any problems.

Then, one snippet assigns **y** and the other assigns **middle**. A possibility is to extract *two* pairs of methods: the first pair of methods computes the value that is assigned to **y** and the second pair of methods computes the value that is assigned to **middle**. Because only the first snippet actually modifies **y**, only the implementation of the corresponding method in the first class is non-trivial. The corresponding method in the second class just returns the **y** argument unmodified. For the same reason, the second method (computing the value of the **middle** variable) only has a non-trivial implementation in the second class: only the second snippet actually modifies the **middle** variable. Nevertheless, the template method needs to call both methods to ensure that both variables are modified as necessary.

After the **y** and **middle** variables have been handled, we again encounter the same variable, **max**, assigned by both snippets, like for the initial assignment to **min**.

We can continue that way up to the end of the lists. We would get only two "matched" variable assignments in this example: **min** (1st one) and **max** (3rd one).

A better approach though, is to first "align" the tokens of the two write-accesses lists in order to maximize the number of matched variable assignments. At the same time this minimizes the number of extracted trivial methods that just return one of their arguments unmodified.

With a proper implementation, we can get three matched variable assignments (shown here in bold) instead of two in our example:

```
[min, y, max, middle, max]
[min,          middle, max, min, y]
```

The resulting template method could then look like the following after all the necessary methods have been extracted and given meaningful names:

```
min = getMin1(x, y);
y = getY1(y);
max = getMax1(x, y, max);
middle = getMiddle(min, max, x, y);
max = getMax2(max, min, middle);
min = getMin2(min);
y = getY2(x, min, y);
```

Listing 5

This is a complex, but correct result, although several extracted methods still have a trivial implementation in either of the two classes. This is the case for **getY1** and **getMax1** in the second class, and **getMin2** and **getY2** in the first class.

We will discuss some subtleties of method extraction more in details in section 5.

How to align the variable assignments so that the number of matches is maximized? Basically, this is an instance of the "Longest Common Subsequence" (LCM) problem with two lists. Hence, we can (again) use a differentiation algorithm, as it precisely solves this problem. The only difference with section 3 is that here we are only interested in the individual pairs of matched tokens, and not in the sequences of *consecutive* pairs of matched tokens. This difference is minor though and does not imply any complication in the implementation.

We still want to point out a potential problem though. A differentiation algorithm, by definition, is an algorithm that solves the "Longest Common Subsequence" problem (and optionally groups the consecutive matched pairs). This problem can be solved in $O(n^2)$ complexity using dynamic programming [18]. Unfortunately, some optimized implementations of the algorithm are using heuristics that may yield to undesired behaviours.

As an example of such a heuristic, we found an implementation based on fast algorithms coming from the field of data compression. While it shows improved performances on various kinds of inputs, it also exhibits a property that is not part of the initial problem: it does not only maximize the number of matched tokens, but also the average length of the sequences of *consecutive* pairs of matched tokens.

This property is not desirable when aligning the two write-accesses lists discussed in this section, because the fact these tokens are consecutive or not is meaningless. On the other hand, this property might be relevant when differentiating statements as we did in section 3: maximizing the length of consecutive matched statements maximizes the average size of the methods to extract, and thus potentially minimizes the number of methods to extract.

We now go back to the final result: the produced template method. Another question one may ask himself is whether the template method is better than the original code. For instance, after the transformation, both classes may contain various trivial methods that just return an argument unmodified. Furthermore, because the extracted methods are requiring different arguments in each class, we have to supply their union so that two corresponding methods have the same signature. As a result, every extracted method potentially has arguments that are not actually used, but are just here to match the signature of the corresponding method in the other class.

In this particular example, our new approach does not give an optimal result. A single method that returns an object or an array containing all the modified values, as suggested in the beginning of this section, would probably be better. But remind that our implementation provides many alternatives to the user, letting to her the choice of the one to apply.

For each alternative, we found various cases in which it produced the best result, which confirms the relevance of introducing our new approach *in addition* to the existing ones.

4.3. Control Flow Breaks

In the previous section, we have dealt with one of the preconditions of the "extract method" refactoring: the fact that a Java method cannot return more than one value. In this section, we investigate another precondition: the extracted method cannot contain a "flow break" [2, 5]. A flow break is a statement that transfers the execution to a point that is no longer reachable when the fragment is extracted in a new method. There are two such statements:

- A **return** statement. If it were extracted in a new method, it would have to be replaced by a statement that escapes *two* methods, which is not possible.
- A **break** statement (extracted without its enclosing block). If it were extracted in a new method, it would have to escape the method *and* the enclosing block in the calling method.

Thrown exceptions on the other hand are not a problem. If the corresponding **catch** block is not in the extracted method, it suffices to declare the exception type as being thrown by the extracted method. Unlike a **return** statement that can only escape a single method, an exception is propagated down the stack until a method catches it. It can thus escape many methods at once.

Our algorithm solves these problems related to flow breaks in a way that is similar to previous research on the C language [12]. The idea is the following: the extracted method has to return an additional "status" value. This value tells the calling method whether it has to issue a **break** or **return** statement that cannot be performed directly be the extracted method itself.

Concretely, a **break** or **return** statement in the extracted method is replaced by a **return** statement with the "status" value. In the calling method, this status value is checked just after the invocation, and the corresponding action is taken: issuing a **break** statement, issuing a **return** statement, or just doing nothing (and continuing the execution flow normally).

The following example illustrates this process. Assume we want to extract the body of the following **while** loop:

```
while (test()) {
  if (x < 0)   // extract from here …
    break;
  else if (x > 0)
    return;
  moreStuff(); // … to here
}
evenMoreStuff();
```

The extracted method returns the status value as a member of an enumeration named **FlowType** in this example:

```
FlowType extracted(int x) {
  if (x < 0)
    return FlowType.BREAK;
  else if (x > 0)
    return FlowType.RETURN;
  moreStuff();
  return FlowType.NORMAL;
}
```

The original method, after the extraction, has to check for the returned value and to issue the "real" **break** or **return** statement as appropriate:

```
while (test()) {
  FlowType status = extracted(x);
  if (status == FlowType.BREAK)
    break;
  else if (status == FlowType.RETURN)
    return;
}
evenMoreStuff();
```

In addition to this scheme that was previously proposed for the C language [12], we have to combine it with the constraint discussed in section 4.2. Indeed, by adding an additional return value (the status) we can easily end up with a method returning more than one value. In practice, our algorithm first handles flow breaks as discussed in this section, and then handles multiple outgoing flows as discussed in section 4.2. Any additional return value generated to resolve flow breaks is then transparently handled when resolving multiple outgoing flows.

4.4. Block Boundary Crossing

As a last precondition of the method extraction process, we cannot extract consecutive statements if they cross the boundary (beginning or end) of a block but do not include the entire block itself. By block, we mean the body of a control statement such as a loop or a conditional.

This precondition is easily resolved, but it can seriously degrade the quality of the overall result. Our algorithm differentiates two cases and takes the following actions.

If the block corresponds to the *same* control statement in both fragments (such as two **while** statements), the two fragments are split in two parts: one part inside the block and the other part outside of the block. Each part can then be extracted separately in its own method. The control statement itself remains in the template method. If the initial fragments cross control expressions (for example the two **while**'s conditions in the case of two **while** loops), we may also need to extract the pair of control

expressions themselves in another pair of methods if they do not match.

If the block does *not* correspond to the same control statement in both fragments, we cannot do any magic. The solution we propose is simply to extend the two fragments in such a way they both cover the entire control statement. We may end up loosing a lot of duplicated statements from the template method. But again, we are at least able to form it.

The same technique has to be used if only one of the two fragments actually contains a control statement.

When we extend a pair of fragments to make them cover an entire control statement, we are adding statements that were not part of them, that is, statements that are *the same* in both methods. Thus, both fragments can be extended in the same way without ambiguity. In case we reach another pair of fragments of *different* statements during the extension, the new pair of fragments is simply merged with the one being extended. Then the extension continues (if necessary) with the new pair.

Finally, there are additional subtleties that must be handled for various specific constructs of the language. As an example, consider the three control expressions of a **for** loop. The second and third control expressions can usually be extracted without any problem in case they differ. The first control expression on the other hand cannot be extracted easily if it contains the declaration of a *different* variable in each of the two fragments. Our implementation just considers that the two loops are not the same in this case, and extends the two fragments to entirely cover them. Other solutions could be possible but are beyond the scope of this paper.

4.5. Further Reducing Duplicated Code

We have presented various ways of modifying the fragments to extract in sections 4.1 to 4.4. These modifications all have the same common goal: making the extraction of the fragments possible. They also have a common drawback though: duplicated statements are included in the extracted methods in some cases.

In some specific situations, the amount of duplicated statements that are added to the extracted methods can be quite large.

We believe this is not necessary a problem in practice: after the template method has been formed, the remaining duplicated statements can be extracted using clone detection and removal techniques if necessary. More generally, by the definition of a refactoring, we can also expect that the user will usually only choose to form a template method when this actually improves the code and significantly reduces the duplicated code. As such, worse-case situations are expected to be rare in practice.

4.6. Other Issues

Classes, methods, fields and their relations form a single graph that is relatively easy to model in an elegant way [6]. On the other hand, dealing with statements as we do is a difficult problem in general. There are a lot of different statements in a language such as Java. Even if some statements are very similar and can be handled in the same way, the number of categories that must be handled differently remains quite large. We will thus not dig into all the details and subtleties that our algorithm has to handle for the numerous special constructs that are, most of the time, very specific to the Java language.

Nevertheless, we would like to point out that we have only considered code statements from the syntactic perspective yet. Our algorithm could be further improved by exploiting the semantics of the statements.

For instance, the differentiation process described in section 3 only detects duplicated statements if they use local variables with the same names. In practice, we would like to also detect duplicated statements in which some variables have been renamed.

There are various other semantics that could be used to improve our algorithm. Most of them have already been covered in the field of clone detection. For example some operators are commutative, and their operands can be swapped safely [16]. Some statements are independent from each other and their execution order can be altered [11]. Finally, conditionals and loops give room for various semantics preserving transformations such as promotion or predicate duplication [12]. All these transformations require complex analyses, but they can potentially increase the quality of the result.

On the other hand, we do not believe that they are of great value for our problem. Recall that forming a template method is a refactoring, that is, a process that is mainly *controlled by the user*. This differs from clone detection, whose purpose is (usually) to *automatically* recover some structure from huge legacy code.

Nevertheless, we want to point out that the high-level structure of the second step of our algorithm can be summarized as follows:

- Apply constraint 1
- Apply constraint 2
- Apply constraint 3
- Etc.

We have presented the most important constraints in sections 4.1 to 4.4. Each constraint has the AST, and a list of duplicated and different code fragments as input. The simplicity of this design makes it very easy for us to add additional constraints if necessary, or to adapt our algorithm for other programming languages.

5. Method Extractions

In this section, we briefly describe the last step of our algorithm, extracting the detected and transformed code fragments into new methods. Because method extraction is already heavily covered in the literature [2, 3, 12], we restrict ourselves to the issues that are specific to our problem.

Without going into the details of method extraction, one of the most difficult problems in the process is to determine the arguments and results of the extracted methods. Our approach is similar to existing ones, but we still have to deal with a notable difference: we have to extract methods by *pairs*, and the two methods of a pair must have the *same signature*. The reason is that both methods must be implementations of the same abstract method that is invocated by the template method.

There are fortunately no major issues in solving this new problem: it suffices to consider the *union* of the required arguments and the *union* of the required results as the actual arguments and results of the two methods.

Obviously, the union of the results is not considered just before we extract the methods, but rather at an earlier stage, when we have to deal with multiple outgoing data flows, as described in section 4.2.

6. Current State

The algorithm described in this paper has been implemented as an Eclipse plugin. The implementation is heavily based on the "**jdt**" (Java Development Tools) library provided by Eclipse. This library already handles the parsing of source code into an AST and the rewriting of an AST into source code.

Our implementation can already process the examples presented in this paper. It has been tested successfully on various other cases coming from real projects. There are some constructs specific to the Java language whose implementations are still in progress though, such as the use of method's local and anonymous classes, which impose some additional restrictions that are not yet handled. A full user interface is still in development stage as well.

Because we are not aware of any previous implementation of the "form template method" refactoring, we can not easily compare our work with previous research. But preliminary tests showed that our implementation already gives correct results on a reasonable subset of the Java language.

Tested on random pairs of methods, the proposed template methods were usually correct but rarely satisfactory. This is not a problem because the user will usually only choose pairs of methods for which the refactoring is a relevant improvement of the program structure. Indeed, when tested on selected pairs of *similar* methods, the proposed results were usually close to our expectations.

The detection part described in section 3 (with the addition of the constraint described in section 4.1) could also be used for clone detection. Preliminary tests showed that our approach was competitive with other approaches, with different qualities. For instance, we did not reach the quality of previous approaches using various semantics information. On the other hand, unlike some of the best approaches, ours did never produce false positives, which greatly simplifies the extraction process. Our approach also gives better results than string-based ones.

The novel scheme presented in section 4.2 is not better than the existing schemes used for method extraction *on average*. Nevertheless we found various *individual* cases in which it yields to the best results. This is usually achieved when used on pairs of methods that perform distinct and unrelated tasks one after the other: if each task manipulates a different variable, they are almost naturally extracted in different methods pairs. We get the worse results with methods that are doing mixed computations on a set of two or more variables that are closely related, such as three coordinates. In such a case, extracting a single method that returns an array is usually the best scheme. These results validate the pertinence of our choice to implement multiple schemes and to present all alternatives to the user.

Finally we even found cases in which our implementation of the *method extraction* part used alone performed better than the corresponding implementations of the Eclipse, NetBeans and Visual Studio development environments. These results are detailed in another paper [1].

7. Related Work

To our best knowledge, the "form template method" refactoring has not been implemented yet in existing development environments. It is a well defined transformation though [9, 10, 14].

A lot of research exists regarding transformations on the high-level structure of a program such as renaming, moving or wrapping fields, classes and methods [3, 6, 7]. Obviously, we need to deal with these aspects when we create a new method and give it a name, but these are beyond the scope of this paper.

Method extraction is an important part of the process of forming a template method. Various researches have been done in this domain [2, 12, 15, 16], and some of them have investigated the Java language. As previously mentioned, method extraction, when considered alone, takes the subset of statements to extract as an immutable input of the algorithm. In our case, we have some freedom in changing the subsets of statements before we extract them in order to improve the overall result. We are not aware of previous work that takes profit of this freedom. This part of the transformation (discussed in section 4) is indeed the main contribution of this paper.

When discussing outgoing data flows, we introduced the notion of variables that are written in a code fragment *and read afterwards*. This concept is far from trivial to implement correctly. We did not go into the details because our solution is entirely based on existing approaches, but the problem is complex and is discussed in various other papers [1, 2, 5].

Clone detection and removal is a problem very similar to the process of forming a template method. In both cases, it is necessary to detect duplicated code statements. The main difference is that clone removal consists in extracting the *duplicated* code statements in new methods, while forming a template method consists in leaving the duplicated code statements, and extracting the *differences*. Various techniques have been investigated to detect clones automatically [8, 13], but only a few authors have investigated the problem of clone extraction [16]. In particular, none of them introduces an additional step explicitly between the detection and the extraction to improve the results.

We suggested various future improvements of our algorithm using semantics in section 4.6. Approaches handling specific cases are discussed in the literature [11, 12, 16], but no general solution has been proposed yet.

8. Conclusion

In this paper we presented a new algorithm that performs a complex refactoring: forming a template method. We showed that the process is close to clone detection and method extraction, but involves some additional difficulties.

We proposed various solutions to the problem and showed that some aspects of our algorithm could also be used to enhance existing tools dealing with clone detection and removal.

We proposed a novel approach by introducing an additional step between the detection of differences and their extractions. This step allows the transformation to be performed even on difficult situations in which it would fail otherwise. We also introduced a new technique to resolve the problem of multiple outgoing data flows (returning more than one result) when extracting a method.

Furthermore we presented a structured implementation in which the steps are cleanly separated from each other, leaving a solid basis for further improvements. We validated and tested our theory by implementing the transformation as an Eclipse plugin and applying it on concrete code samples.

References

[1] Nicolas Juillerat, Béat Hirsbrunner: *Improving Method Extraction*, 1st Workshop on Refactoring Tools, TU Berlin Technical Report, ISSN 1436-9915, pp. 48 – 49, 2007.

[2] Mathieu Verbaere, Ran Ettinger and Oege de Moor: *JunGL: a Scripting Language for Refactoring*, 28th International Conference on Software Engineering, pp. 172 – 181, 2006.

[3] Leif Frenzel: *The Language Toolkit: An API for Automated Refactoring in Eclipse-based IDEs*, Eclipse Magazin, vol. 5, 2006.

[4] Nicolas Juillerat, Béat Hirsbrunner: *An Algorithm for Detecting and Removing Clones in Java Code*, Proc. of the 3rd Workshop on Software Evolution through Transformations, pp. 63 – 74, 2006.

[5] Nicolas Juillerat, Béat Hirsbrunner: *FOOD: An Intermediate Model for Automated Refactoring*, 5th International Conference on Software Methodologies, Tools and Techniques, pp. 452 – 461, 2006.

[6] Tom Mens: *On the Use of Graph Transformations for Model Refactoring*, International Summer School on Generative and Transformational Techniques in Software Engineering, pp. 67 – 98, 2005.

[7] Günter Kniesel: *ConTraCT - A Refactoring Editor based on Composable Conditional Program Transformations*, International Summer School on Generative and Transformational Techniques in Software Engineering, pp. 79 – 93, 2005.

[8] Tom Copeland: *PMD Applied*, Centennial Books Online, 2005.

[9] Joshua Kerievsky, *Refactoring to Patterns*, Addison-Wesley, 2004.

[10] Tom Mens, Tom Tourwé: *A Survey of Software Refactoring*, IEEE Transactions on Software Engineering, vol. 30, no. 2, pp. 126 – 139, 2004.

[11] David Koes, Mihai Budiu, Girish Venkataramani: *Programmer Specified Pointer Independence*, Proceedings of the 2004 Workshop on Memory System Performance, pp. 51 – 59, 2004.

[12] Raghavan Komondoor, Susan Horwitz: *Effective, Automatic Procedure Extraction*, 11th IEEE International Workshop on Program Comprehension, pp. 33 – 42, 2003.

[13] Elizabeth Burd, John Bailey: *Evaluating Clone Detection Tools for Use during Preventative Maintenance*, Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation, pp. 36 – 43, 2002.

[14] Martin Fowler: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2002.

[15] Magdalena Balazinska et al.: *Partial Redesign of Java Software Systems Based on Clone Analysis*, Proceedings of the 6th Working Conference on Reverse Engineering, pp. 326 – 336, 1999.

[16] Ira D. Baxter et al.: *Clone Detection Using Abstract Syntax Trees*, IEEE Proceedings of the International Conference on Software Maintenance, pp. 368 – 377, 1998.

[17] Mark Nelson, Jean-Loup Gailly: *The Data Compression Book*, M&T Books, 2nd edition, 1995.

[18] J. W. Hunt, M. Douglas McIlroy: *An Algorithm for Differential File Comparison*, Bell Laboratories Computing Science Technical Report 41 (available on the home page of the 2nd author), 1976.