AUTOMATIC DETECTION OF POSSIBLE REFACTORINGS

Stasys Peldzius
Vilnius University, Faculty of Mathematics and Informatics, Naugarduko str. 24, LT-03225
Vilnius, Lithuania, stasys.peldzius@mif.vu.lt

Abstract. In the continual evolution of software applications should be ensured that these are high quality designed and programmed. But inevitably appear the defective code that call “code smell”. It is therefore important to be able to find such problems, and to correct it. Code refactoring is a process which solves “code smell” and to improve maintainability, performance, or to improve extensibility. The aim of this paper is to investigate and propose automatic – universal refactoring tool, which is detected in self-“code smell” and is independent of software systems programming languages. This tool uses a logic programming, which is a factual description of the conversion, and the rules – the refactoring of the program. It also aims to create a practical benefit of the automatic adjustments to the proposed tool is to be realized, and a demonstration of refactoring operation.

Keywords: refactoring, logic programming, code smell, software evolution.

1 Introduction

The development of the software applications or their subsequent improvement very often result in the reduction of their quality, creation of over-large classes, too long methods [6], and the class hierarchy fails to meet the requirements, so programmers have to solve these problems by resorting to the refactoring process. The refactoring is the process of changing application’s internal structure without modifying its existing functionality [5]. Refactoring improve the readability and enable easier extensibility of the programs.

The refactoring is used by all the programmers, but usually they are performed manually, thus the programmers themselves have to detect the location of the bad smells (any symptom in the source code of a program that possibly indicates a deeper problem) and to perform an appropriate refactoring. Usually it is a very tedious task that could become much easier having a tool that would help identify the places that need to be refactored. Although such tools have already been developed, they can refactor only those places of the program that are identified by the programmer, and can only perform simple refactoring, such as detecting declared but unused variables. However, the programmer usually faces a contrary problem – the tedious task is not to perform a refactoring but to detect a bad smell and there are no tools developed for such a purpose. The major advances have been made in the application of logic programming for solving this problem [16]. This paper offers a model for creating a program that could perform refactoring automatically and serve as a universal refactoring tool which could be used by programmers in practice; the paper also presents detailed examples of its usage.

2 Refactoring Tools

There is a diversity of refactoring tools available, but in practice they usually do not meet any of these requirements and can only detect the places that are in need of simple refactoring, such as declared but unused variables, and do not locate the places that need a more complex refactoring. Another disadvantage of such programs is that the user is either not able to develop new refactoring for these tools or it is an overly complex task which requires specific knowledge and comprehension of the tool structure; in addition, they are developed for the programs that support a specific programming language and rewriting it for another language can be complex and expensive. While scientific studies [3, 4, 7, 10, 11, 12, 14, and 15] present various ways for refactoring automation and offer theoretical models for performing wider range of refactorings, such as [1] duplicate code detection using anti-unification, it is difficult to apply them in practice since each new refactoring requires a creation of a complex model. Present [17] three distinct steps in the refactoring process:

1) detect when an application should be refactored,
2) identify which refactoring(s) should be applied and
3) where (automatically) perform these refactorings.

The first step depends on the programmer and the second and third can be automated in the tool. The model of the tool presented in this paper must meet additional requirements which are useful for the programmers:

1. Automatic detection of possible refactorings. It is the most important requirement since the programmer does not have to look for the places that do not satisfy the requirements, so this tool is expected to detect the parts which need to be refactored.
2. A possibility to expand the number of refactorings in the tool in a simple way. These are important refactorings which occur during the development of specific systems or performing it in a specific way, i. e. each
team of the programmers may need to define their own refactoring instead of hoping that that the suppliers of the tool will update the refactorings supported by the tool.

3. Easy adaptation of the tool to various programming languages; it is required that the refactoring written for one language could be easily modified for another language.

The logic meta-programming is used to detect the places that are in need of refactoring since this language can be used in writing declarative programs which describe the examined programs and manipulate them. The practical application of this program is demonstrated in various scientific studies [2, 16, 17, 18, and 19] which include experiments with the Smalltalk and Java languages; however, they do not fulfill the stated goals. This paper aims at expanding their ideas and presents a model of the universal tool which serves the 3 goals defined above. The presented tool is related only to the logic programming and can be integrated with any language in which the analyzed programs are written. Furthermore, the paper analyzes the realization possibilities of such a tool and presents examples that examine the solutions of several important problems which reduce the quality of the programs.

### 3 Language of Refactoring

The language of refactoring is not only used to write the algorithms for the detection of the possible refactorings, but also to define the refactored program since the language-independent refactorings can only be implemented by having an intermediate form where programs supporting different languages could be entered. If this intermediate language is uniform for all languages, it can be called a universal language. In this case the refactorings will be searched not in the primary application, but in the intermediate form which defines the refactorings data. This language of refactoring will be used for defining the refactorings data and getting the results. The Prolog logic programming language was selected as a refactoring language.

#### 3.1 Refactorings Data

The refactorings data is the information about the analyzed program, for example, the names of the classes, methods, relationships between them etc. The refactorings data are obtained by transforming refactored programs, acquiring the information necessary for performing a desired refactoring. The refactorings data are defined by the Prolog facts. Theoretically, it is possible to rewrite the entire program by facts as shown in Table 1. However, depending on the refactoring, certain data have to be specified.

**Table 1. Rewriting the program by facts [18]**

<table>
<thead>
<tr>
<th>Object – oriented program</th>
<th>Logic program facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Stack { int pos = 0; public Object peek ( ) { return contents[pos]; } public Object pop ( ) { return contents[--pop]; }</td>
<td>Class(,,Stack”). Var(,,Stack”, ,,int”, ,,pos”). Method(,,Stack”, ,,Object”, ,,peek“ [, ] [return contents[pos]]). Method(,,Stack”, ,,Object”, ,,pop” [, ] [return contents[--pop]]).</td>
</tr>
</tbody>
</table>

During the application transformation it is important not only to pass over the data about the application structure, but also about the relationships within that structure, the data should form a large connected graph with clear connections. This example shows how the required information about the application structure (classes, methods, variables) and relationships (method, whether the variable belongs to some class) can be transferred. Also any object-oriented language could be rewritten by these facts since they are not tied to specific key words, but are related to the language features (class, interface, method, variable). A different refactoring list could be created for the structured programming languages (procedures, functions) and other groups of the programming languages since the refactorings depend on these groups. Certain refactorings which are important for the object-oriented programs may not be adapted to the structured programs and vice versa. The standard refactorings data about the applications are shown in Table 2.

**Table 2. The examples of the data used in refactoring operations [13]**

<table>
<thead>
<tr>
<th>Representational Mapping Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>class(C)</td>
<td>C must be a class</td>
</tr>
<tr>
<td>subclass(P, C)</td>
<td>class C must be a direct subclass of class P</td>
</tr>
<tr>
<td>concreteSubclass(P, C)</td>
<td>class C must be a concrete subclass of class P</td>
</tr>
<tr>
<td>abstractMethod(C, M)</td>
<td>M must be an abstract method of class C</td>
</tr>
<tr>
<td>concreteMethod(C, M, B)</td>
<td>M must be a concrete method with body B in class C</td>
</tr>
</tbody>
</table>
Representational Mapping Predicate | Description
---|---
classMethod(C, M, B) | M must be a class method with body B in class C
instanceVariable(C, V) | V must be an instance variable of class C
objectCreationBody(M, B, C) | body B of method M must create an instance of C

These data contain primary information about the program that could not be obtained from inferences. By further manipulating them, it is possible to get various information about the application itself, for instance, to determine whether a class contains at least one method, the number of subclasses of the class etc. By combining these data it is possible to define the refactoring program which while analyzing the data will inform if the program needs to be refactored.

### 3.2 Refactorings Programs

The refactorings programs are defined by means of logical rules. It is rather easy to write the refactoring program in the logic language – you simply need to write a syntactically correct task. For example, there is a following refactoring rule: a class needs to be split into two classes (abstract and concrete) if the figure calculated by dividing the number of abstract methods by the total number of the methods within that class is less than a half and does not equal zero (meaning, that there are no abstract methods). After formulating the task in such a way, it can be implemented word by word and the resulting refactoring program is written by Prolog facts as shown in Table 3.

#### Table 3. Classes that are in need of refactoring

<table>
<thead>
<tr>
<th>Refactoring data (Prolog facts):</th>
<th>Refactoring program (Prolog rules):</th>
<th>Refactoring result (Prolog answer):</th>
</tr>
</thead>
<tbody>
<tr>
<td>class(a); class(b); method(a, x, true); method(a, y, false); method(a, w, false); method(a, x, false); method(b, xx, true); method(b, yy, true); method(b, ww, true); method(b, xx, false);</td>
<td>abstractRatio(C,N) :- class(C), findall(C,method(C,<em>,true),List1), countList(List1,A), findall(C,method(C,</em>,_),List2), countList(List2,B), N is A / B, N &lt; 0.5, N ==0.</td>
<td>Class = a, Ratio = 0.25 ; Class = b, Ratio = 0.75 ; false.</td>
</tr>
</tbody>
</table>

**Explanation:**

From this answer it can be implied that the class “a” has less than a half abstract methods (0.25) so it needs to be split into two classes that share its abstract and simple methods. The class “b” has more than a half abstract methods so it can remain unchanged.

This program applies a standard Prolog rule: `findall`, returning a set of all facts that satisfy the rule. The rule applied for calculation of set members can be implemented in this way:

```prolog
countList([],0).
countList([_|Tail],N) :- countList(Tail,N1), N is N1+1.
```

It provides a simple way to implement the refactoring which automatically detects all classes that do not meet the abstractness requirement if such a requirement is raised. In order to achieve this, only two types of data are needed: data about the class and the abstractness of their methods. First of all a standard base of facts used for refactoring can be created and constantly supplemented while developing new refactoring program. In this way the creation of new refactoring program will become increasingly easy since a large database will be developed.

### 3.3 Refactorings Results

The results of the refactoring have to satisfy the refactoring task. The refactoring can be written either for informing the programmer about the locations where refactoring has to be performed or to enable the refactoring tool not only to detect the places that need to be refactored but also to perform them by rearranging the source program. In either case, the result of the program has to be interpreted in the same way. Usually the refactoring tool detects the places that need refactoring and the programmer takes care of the implementation of this operation. For example, the refactoring program can detect all classes that are overly-large (according to a
certain number of sentences) but it will not able to split them. However, the tool is able not only to detect the unused methods or variables, but also to delete them.

4 Universal Refactoring Tool

The defined language of refactoring still lacks a way to obtain the data from the refactored program and a means to interpret the results. The part of the tool that will get the data from the program for performing the desired refactoring will be called the data generator. The part of the tool responsible for interpreting the results will be called the results interpreter. The model of the tool is presented in Figure 1.

![Figure 1. The model of the Universal Refactoring Tool](image)

These parts of the program should be programmed in the same programming language as the examined program, since it is impossible to write a universal program that could get the information about the programs written in any language. This requirement corresponds to the stated universality goal since it is important that the refactorings programs would be language-independent and written in a logic programming language.

4.1 Data Generator

Data generator is a program that analyzes the refactored program and gets the information from that program needed to perform the refactoring. The majority of the programming languages (C#, Java, PHP etc.) have Reflection libraries that provide the data describing the program’s structure. For example, the fact for getting class information is presented in Table 4.

<table>
<thead>
<tr>
<th>C# method</th>
<th>JAVA method</th>
</tr>
</thead>
<tbody>
<tr>
<td>using System.Reflection;</td>
<td>import java.lang.reflect.*;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>public void AllClass(Assembly assembly)</td>
<td>public void AllClass(Class[] classes)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>Type[] types = assembly.GetTypes();</td>
<td>for (Class c : classes)</td>
</tr>
<tr>
<td>foreach (Type type in types)</td>
<td>System.out.println(&quot;class(&quot; + c.getName() + &quot;);&quot;);</td>
</tr>
<tr>
<td>{</td>
<td>}</td>
</tr>
<tr>
<td>Console.WriteLine(&quot;class(&quot; + type.FullName + &quot;)&quot;);</td>
<td>}</td>
</tr>
</tbody>
</table>

Table 4. Extraction of the Class(Name) fact
This part of the tool creates all the data necessary to perform refactoring and later can be adapted to any other program, written in that programming language.

4.2 Results Interpreter

The results interpreter is a program that performs operations with refactoring results. This program is separately designed for each programming language if there is a need to make rearrangements in the source program. If it is sufficient to show the refactoring result to the programmer, it is possible to develop a universal program that would transform the results of the Prolog rules into those acceptable to the programmer. There are two ways that the results interpreter can rearrange the results, presented in the Table 3, firstly to write the explanation of the results or to implement the modifications within the program. If there is a need to perform the modifications in the program automatically, the most convenient implementation would be the integrated development environment (IDE) plug-ins. There is no need to create a new program since it is possible to use the existing refactorings programs which are not able to detect the flaws in the program but can perform the refactoring which is identified by the programmer while the programmer will be informed by the result of the logic refactoring program. This implementation is the most challenging part of the tool. In most cases it is sufficient to provide an informative answer to the programmer by identifying the place which needs to be refactored, and even to offer the way to perform this operation, but leaving to carry out the refactoring to the programmer.

The following chapter presents the example of the refactoring. A detailed analysis is carried out as to what primary data have to be collected, how to write a refactoring program and how to understand the obtained results.

5 Acyclic Dependencies Refactoring

The term package is used in Java languages. In this language a package is used to collect a logical grouping of declarations that can be imported into other programs. In Java, for example, one can write several classes and add them into the one package. Then other Java programs can use that package and to gain access to those classes [8]. In Microsoft .NET Framework package match library assembly.

The relationships between the packages provide very important information about the system. The design of the systems usually starts from the package relationships, then it proceeds down to the package level and then the specification of classes and their relationships is made. A well-designed project contains packages that are connected by the acyclic directed graph. The example analyzed in this chapter is shown in Figure 2.

![Figure 2. A directed acyclic graph [8]](image)

Given such kind of hierarchy, each package can be changed since it is clearly known, which packages are dependent upon it and which are not. For example, if the package MyDialogs is changed, it is clear that the packages MyTasks and MyApplication will also need to be checked. All other packages are not interested whether MyDialogs is changed or not because they do not have any relationship with it. However, if there is a need to change one class of the MyDialogs package and within this class the method of the MyApplication class must be used, then the relationships between packages become interdependent as shown in Figure 3.
This modification creates a cycle between packages. Such cycles between packages cause major problems. The change of the package within the cycle can lead to unexpected consequences if the programmer is not aware of this cycle. Therefore, when there is a cycle, the changes must be made to all packages if one package needs to be changed. It can be said that the packages within the cycle turn into one large package. Therefore, it is important to avoid such cycles and this can be achieved easily. What you need to do is simply move the method, which is used within the cycle, to the abstract class, then inherit this class, and the class within the cycle simply establishes a relationship with the abstract class which belongs to the same package.

In order to solve this problem it is possible to formulate a refactoring task: detect the package cycles and determine in which of the packages the cycle needs to be broken. This can be done by measuring stability of the package [9]: $I = \frac{Ce}{(Ca + Ce)}$, where $Ce$ indicates the number of classes inside this package that depend upon the outside packages, $Ca$ – the number of classes outside this package that depend upon the classes within this package. When the $I$ metric is 0, it indicates a maximally stable package because it does not have relationships with the outside classes. When the $I$ metric is 1, it indicates a maximally unstable package as no other package depends upon this package but it calls other packages. The more stable the package is the more abstract classes it has to contain; when the $I$ metric is 0, it means that the package contains nothing but abstract classes. This metric could be used when eliminating the cycles within packages. The most stable class must be found for inheriting from the abstract class that would have the methods needed for the class that calls the abstract class. In this way the cycle will be eliminated. Having formulated the refactoring task, now we need to create refactorings data. Firstly, all packages and relationships between the packages are defined:

```
package(myApp).  //MyApplication
package(myTask). //MyTasks
package(myDial).  //MyDialogs
package(taskWin). //TaskWindow
package(win).     //Windows
packageUsed( myApp, [taskWin,myTask] ).
packageUsed( taskWin, [win] ).
packageUsed( myTask, [myDial] ).
packageUsed( win, [] ).
```

It is not difficult to obtain these data from the refactored program; therefore, the universal tool would easily write the data generator. The most popular object-oriented languages offer Reflection libraries that provide such information. It is sufficient to have these two types of data in order to perform this refactoring. The refactoring program cycles( Package, Reference) would look like this:

```
cycles(X,P) :- cycles(X, X, P,[]).
cycles( X, X, [] ) :- member(X,A),!.
cycles( X,Y,P,A ) :-
  packageUsed( Y, ArcList ),
  member( Z, ArcList ),
  not(member( Z, A)),
  P = [Z|PTail],
```

In order to use the metric that would calculate the stability of the classes, additional data would be needed, inserting information about packages (Package, Ca,Ce):

```
packageUsed( myApp,3,1 ).
```
packageUsed( taskWin, 1,1 ).
packageUsed( myTask, 1,2 ).
packageUsed( myDial, 2,1 ).
packageUsed( win, 0,2 ).

Also an additional rule should be created that would calculate the metric value:

\[
\text{packageStability}(P, I) \leftarrow \text{packageUsed}(P, Ca, Ce),
I \text{ is } Ce / (Ca+Ce).
\]

And supplement the main refactoring program:


Now the programmer is aware for which package an abstract class needs to be created in order to eliminate the cycle. This refactoring program can be further extended by incorporating into the refactorings data not only the number of the package relationships but also by specifying which classes call which methods. Having these data it is possible to expand the refactoring so that after detecting the refactored package it would specify which methods need to be implemented after inheriting the abstract classes, in other words, which methods within this package are used by other packages that create cycles. The information required for the expansion of the refactoring is presented as the following data:

packageRef( myApp, app, calc, myDial).
packageRef( myTask, task, list, myApp).
packageRef( myTask, task, count, myApp).
packageRef( myDial, dial, div, myTask).
packageRef( taskWin, win, form, myApp).
packageRef( win, window, visible, taskWin).
packageRef( win, window, enable, myDial).

Thus having a program that detects cycles and having the data about the relationships, it is possible to write a program that could take out all the methods which can be moved to the abstract class as demonstrated in Table 5.

Table 5. Refactoring result

<table>
<thead>
<tr>
<th>Refactoring program:</th>
<th>Refactoring result:</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract(P,C,M,K,I) :- cycles(P,L,I), packageRef(P,C,M,K), member(K,L).</td>
<td>PFrom = myApp, Class = app, Method = calc, PTo = myDial, I = 0.25 ; PFrom = myTask, Class = task, Method = list, PTo = myApp, I = 0.666667 ; PFrom = myTask, Class = task, Method = count, PTo = myApp, I = 0.666667 ; PFrom = myDial, Class = dial, Method = div, PTo = myTask, I = 0.333333 ; false.</td>
</tr>
</tbody>
</table>

This result demonstrates how the cycle emerges. It is advisable to break the cycle by moving the method, existing within the package with the smallest stability ratio, to the abstract class. This information is sufficient to be able to create the results interpreter. It is known, which method needs to be moved to the abstract class and also in which package the relationship must be changed from the former class to the abstract class so that only needed methods would be used, in such a way avoiding the cycle. Such results interpreter would able not only to inform the programmer about the presence of the cycles but also to eliminate them automatically.

This example demonstrates a simple way to solve a complex problem which can highly reduce the quality of the programs. This refactoring program is not tied to the Java language; the same program could support the .NET Framework languages. It could be achieved by re-writing the data generator and the results interpreter.

6 Conclusions

This paper presents a model for performing automated detection of bad smells within the programs and offers a tool that helps apply this model in practice. This model meets the stated requirements – it automatically detects bad smells, allows a simple creation of new refactoring, and implements these refactorings regardless of the programming language used. Another advantage of creating refactoring by means of logic programming is that many programmers are familiar with it and it is easy to learn – it is sufficient to understand the refactoring logics and then to write the program without any difficulties.

It is possible to create a universal refactoring tool that would assist in implementing the refactorings programs. The utilization of such tool would be very convenient and easy and would not require special
resources from the programmer. It would only require writing a data generator that would analyze the programs and select the information necessary for performing refactoring. The functions of the results interpreter could be limited to the presentation of information without modifying it and thus allowing the programmer to decide how to change the program in order to eliminate the problem.

Having a wide range of refactorings programs, the programmer is able to perform all the implemented refactorings which could be adapted to the analyzed system of programs. The programmer will only need to decide whether he wants to refactor the programs according to the obtained result.

References