Java Code Cleanup using ExtendJ

Hannes Jönsson
LTH
stv10hjo@student.lu.se

Felix Olsson
LTH
dat12fol@student.lu.se

Abstract

In order to reduce code size, ensure proper functionality of overridden methods as well as improve clarity of which imports are actually used, a code cleanup tool can be used. This report describes the implementation of such a tool based on the ExtendJ Java compiler. The tool is compared to other tools with similar functionality and its functions are evaluated. The conclusions we draw is that the tool does not produce correct results under all circumstances, a result of the difficulty to ensure coverage of all possible Java functionality.

Keywords Java compiler, code cleanup, imports

1. Introduction

In this report the implementation of source code transformation tool based on the ExtendJ Java compiler is described. ExtendJ is an extensible, modular compiler, built with the use of a declarative attribute grammar system called JastAdd. Its modular design allows developers and users to add new modules, providing the compiler with new features according the the demand of the situation at hand.

During this project we take advantage of this extensible, modular design to add a few useful code cleanup features to the compiler. The main features we focus on concern imports. To be specific, we want functionality that handles unused imports and the possibility to automatically replace on-demand imports with the specific import path. In addition to the cleanup of imports, the ability to automatically place override annotation at appropriate places is added as a feature.

2. Motivating examples

To provide the reader with a better understanding of the goals of the project, this section provides a few examples to better explain the motivations behind the choices of features and implementation thereof in the cleanup tool.

2.1 Import cleanup

When writing code it is common to work in an iterative manner, going back and reworking earlier code, updating it and fixing bugs. This is especially true for certain development paradigms, such as agile development[1]. While doing so, it is unavoidable to change parts of the code. Some of these parts might use an import that you have provided which suddenly no longer is necessary. What we want is an extension for ExtendJ that provides automatic detection of such an unused import, and the removal thereof. As an example, we want to turn this:

```
import java.util.List;
import java.util.HashSet;
import java.io.File;

public class UnusedImports {
  List<Integer> list;
}
```

Code example 1. HashSet is an unused import.

Into this:

```
import java.util.List;

public class UnusedImports {
  List<Integer> list;
}
```

Code example 2. Unused import HashSet is removed.

As can be seen from this simple example, such a feature can increase readability by removing redundant lines from the code.
Another feature which might help to clean up the code would be automatic replacement of on-demand imports with the proper, complete imports according to that which is used in the code. By avoiding on-demand imports, the code becomes clearer and lets the user/programmer directly see which packages and which classes are being used. By transforming code in example 3 into the code seen in example 4, you can avoid ambiguity and possible conflicts.

```
import java.util.*;

public class UnusedImports {
  List<Integer> list;
}
```

Code example 3. Only List is actually used in java.util.

```
import java.util.List;

public class UnusedImports {
  List<Integer> list;
}
```

Code example 4. The on-demand import replaced with the direct import of List.

2.2 Override annotation

At the core of object oriented programming is the possibility to extend a super-class in order to specialize its behavior. This specialization is accomplished through overriding; the specialized re-

[1] K. Beck, Extreme Programming Explained, p28

implementation of methods. There is of course a dependency between a super-class and a sub-class extending it. In the case of a change in the name of a method which is overridden in the extending sub-class, the change must also take place in that sub-class, lest the compiler will not recognize the method as overriding an existing method in the super-class but rather a new method. To assure that the method in question actually overrides a method in the super-class, or, in the case of an abstract method, actually implements it, the override annotation $@Override$ is used. With the override annotation in place an error is produced should the method being overridden not be found in a super-class. A tool which automatically adds this annotation whenever it detects a method which also exists in a super-class, in order to prevent such errors during re-factoring, is thus very useful.

```
public class Super{

public void refactoredMethod(){...}
}

public class SubClass extends Super{

public void method(){...}
}
```

Code example 5. Method in the SubClass is recognized as a new method and not as overriding refactoredMethod.

```
public class Super{

public void refactoredMethod(){...}
}

public class SubClass extends Super{
@Override
public void method(){...}
}
```

Code example 6. Override annotation causes error to be produced.

While comparable solutions exist, for example built in to most IDE's such as Eclipse or IntelliJ, we want an extension to the compiler itself enabling a user access to this feature without having to use a software suite separated from the compiler . We will compare our solution with software providing similar functionality, comparing the size of the software as well as correctness.

3. Implementation

The main way in which we extend the ExtendJ compiler with new functionality is with the use of aspects. Aspects are a way of adding attributes to the components which make up a program [2]. A program can be represented as a tree of nodes called an abstract syntax tree. The nodes of this tree represent different parts of the program. With the use of aspects, we can add attributes to these nodes. As an example, a node can represent the declaration of a method, and be connected with other nodes representing the return type, the name of the method and so forth. With aspects we have a way of adding further attributes to these nodes, without actually modifying the class describing the node itself. Instead, it is done in a completely self-contained, modular way. Such an added attribute can take on many forms; it might for example be a collection of the names of all the methods in a class, stored in an attribute in the node representing the class itself. For further information about aspects and other related concepts see http://jastadd.org/web/documentation/concept-overview.php.

[2] G. Kiczales et al. Aspect-oriented programming, p. 222

3.1 Unused imports removal

The basic idea for the implementation of the removal of unused imports is based around a collection of all types of objects used in the class to be cleaned, as well as a collection of all imported types in the same class. These are stored as collection attributes in the node *CompilationUnit*, which is a node of the abstract syntax tree, representing a Java source file. Any type contained in the set of imported types but not in the set of used types is considered unused, and the line at which this type is imported added to the set *killableLines*.

This code examples shows how the declaration of a variable contributes the type used to the set of used types:

```
VarDeclStmt contributes
    localVariableTypeName()
    when getTypeAccess().nodeType().equals(
        "TypeAccess")
    to CompilationUnit.usedTypes()
    for compilationUnit();
```

Code example 7. The contribution of relevant data to the collection.

A set of Integers called *killableLines* representing the lines where an unused import is located in the source file is populated by comparing the two sets of types.

3.2 On-demand import replacement

The replacement of on-demand imports with the actual path to the type being used in the class is achieved by a method similar to that of the removal of unused imports. The same set of used types is traversed, and for every used type a check is made to see if it is a part of a package imported on demand. In this example

```
import java.util.*;

public class UnusedImports {
  List<Integer> list;
}
```

Code example 8. Only List is actually used in java.util.

The program checks, for the used type List, if it is present in the package $java.util$ which is imported on demand. as this is the case, it replaces $java.util.*$ with $java.util.List$ as follows:

```
import java.util.List;

public class UnusedImports {
  List<Integer> list;
}
```

Code example 9. The wildcard is replaced with the direct import.

This new import is added to a set *importTypesOnDemand*, while the number of the line is added to a Integer set *onDemLines*. A check is also made to see that the import is not already done. A code example provides some insight in the workings of the aspect:

```
eq StaticImportOnDemandDecl.
    isImportedAlready(String name) {
    for(SingleStaticImportDecl s :
        compilationUnit().staticImports())
        {
        if(s.getID().equals(name)){
            return true;
        }
    }
    return false;
}
```

Code example 10. Check to ensure that duplicate imports are avoided.

3.3 Automatic insertion of override annotation

To implement the override annotation feature, collections are once again used. Each *CompilationUnit* holds two sets, one set *methods* which holds the *MethodDecl* objects representing the methods contained in the current class, and a set *overrideMethods* which, if the current class extends a super-class, contains the methods in the super-class. The *method* set is traversed and the a check is made to see if the methods override any method in the super-class, and if so if there is a *@Override* present at that method. If no such annotation is present, the line number where it should be added is put into a collection of integers, *methodLinesWhereWeShouldAddOverrideAnnotation*. The following code shows this calculation of lines where the notation should be added:

```
syn HashSet<Integer> CompilationUnit.
    methodLines...() {
    HashSet<Integer> temp = new HashSet<
        Integer>();
    for(MethodDecl m : methods()){
        for(MethodDecl sm :
            superClassMethods()){
            if(m.overrides(sm) && !m.
                hasOverride()){
                temp.add(m.lineNumber());
            }
        }
    }
    return temp;
    }
```

Code example 11. Calculation of which lines need @Override.

3.4 Cleaning of the source code

Once the static analysis is completed, and all the relevant attributes have been calculated, it is possible to use the results to conduct the actual cleaning of the source code. The source file is read line by line, and each line checked against data sets containing numbers of which lines are to be modified. If a line is marked for modification, the tool carries this out according to data stored in relevant attributes and collections. Each line is then, whether modified or not, written to a output file.

4. Evaluation

To evaluate the implementation, three different evaluation frameworks are used. These frameworks are correctness, size of the implementation and usability. Related works to which our tool can be compared are presented in the section below.

4.1 Related work

In order to provide a context for the evaluation frameworks, related tools against which a comparison can be made must be selected. Many of the tools encountered, such as $Importscrubber$, dates back to the early 00's, often with the latest update as far back as in 2002. As a result, $Importscrubber$ only supports Java 1.4. With the availability of IDE's for Java, this is not a surprise. Eclipse, for example, has a cleanup feature containing all the features provided by our tool, except for the changing of on-demand imports into direct imports. The replacement of on demand imports can however still be conducted by using the import organizer.

4.1.1 Importscrubber

$Importscrubber$ is a stand-alone tool used for cleaning up imports. $Importscrubber$ does mostly everything our tool does, with a few exceptions. For example, it does not handle the use of static constants[3]. Neither does it insert override annotations. Additionally it is limited to Java 1.4.3[4], which may make it impossible to run on a program written with the use of a modern Java version. Thus we never got $Importscrubber$ to work with out test files.

4.1.2 Eclipse

Eclipse is a well known IDE with lots of different functions. Among them are code cleanup tools. Eclipse handles the same functionality as our tool, but it differentiates between cleanup and organization. What this means is that while the cleanup tool removes unused imports and adds override annotations, it does not replace on-demand imports. The replacement of on-demand imports is instead handled by a tool called *import organizer*.

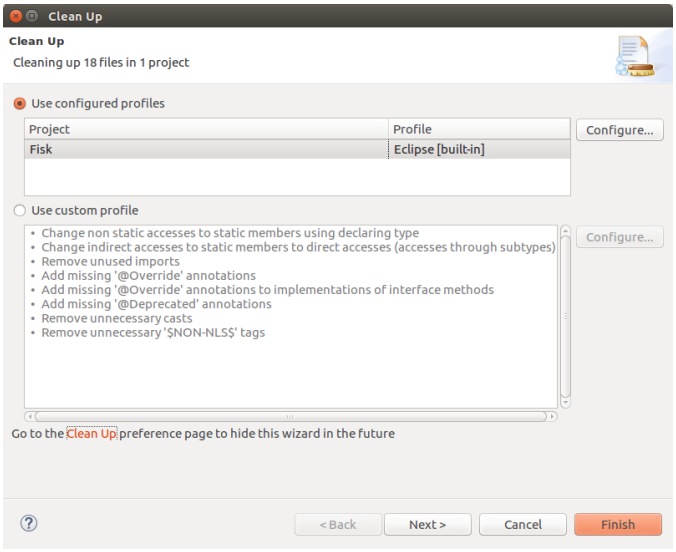

Figure 1. The code cleanup tool in Eclipse.

4.2 Correctness

The first part of the evaluation assesses the correctness of the implementation. This is done by testing the tool on a number of test programs selected from GitHub. We compile the test programs both with and without using our added features. The tool achieves correctness if the resulting program clears the same tests as the program compiled without transforming the source code. In the case of a test program lacking tests, correctness is instead defined by the $javac$ compiler compiling the file without error both before and after the running of our tool.

The first program used to evaluate to the tool is a small, simple version of the game "Snake" found as an open source project on github[5]. Though lacking in unit tests, it does contain a variety of types of imports and uses of statically imported classes and constants. A few examples:

```
import java.util.ArrayList;
import java.awt.Color;

        C.add(Color.darkGray);
```

[3] http://importscrubber.sourceforge.net/limitations.html

[4] http://importscrubber.sourceforge.net/

[5] https://github.com/overben/Snake

```
        C.add(Color.BLUE);
        C.add(Color.white);
        square = new SquarePanel(C.get(color));

}...{
import java.awt.event.KeyListener;

        this.addKeyListener((KeyListener) new
            KeyboardListener());
```

Code example 12. Constants and type casting.

The first example shows the need for the tool to handle constants in an imported class correctly. The second example is especially interesting, since the use of a cast type requires particular attention in the implementation, lest it be handled incorrectly. Neither of these two examples show any code that will be changed, but they are both still of interest since to fulfill the correctness requirement the tool needs to leave code that already is clean as it is. It is important to make sure that imports are not marked as unused when they are used only as a cast, as in the second example. The tool handled these, and all other cases of the first evaluation program, correctly.

The second program used to evaluate the tool is an example implementation of several algorithms together with corresponding unit tests[6]. After running our tool on the source code files of this program and then running the unit tests, some problems were encountered. The tool has a problem recognizing several used types as such, and thus marks the import of used types as unused and deletes them. The deletion of imports that are actually used, though not recognized by the tool, causes the program to break. An example of this is when an interface is instantiated as a concrete class.

```
import java.util.ArrayList;
import java.util.List;

public class ListerOfThings{
        List<String> list = new ArrayList<
            String>();
}
```

Code example 13. ArrayList is not recognized as a used type.

The current iteration of our tool lacks functionality which takes care of the right hand side of such an assign statement as in the example above, causing only *List* to be recognized as a used type. This results in the incorrect removal of the *ArrayList*-import.

Another failure of our tool is in the case of an inheritance chain. That is, if a class C extends class B which in turn extends class A, our program does not recognize a method inherited to C from A as such, and thus removes the override annotation at such a method.

4.3 Size of implementation

The the size of a program without any context is just an arbitrary number without any particular meaning. As such, we will compare the size of our implementation with another tool with similar features to our own.

$importscrubber$ contains 1641 source lines of code[7], SLOC , excluding comments, while our tool is implemented with 446 SLOC. An important thing to consider when looking at these numbers is the fact that $importscrubber$ contains other auxiliary functionality such as the parsing of the files, something which is handled by ExtendJ in the case of our tool. Such a comparison thus easily gives a skewed picture, as it is a comparison of a stand-alone tool and an extension to a compiler.

[6] https://github.com/spinfo/java

[7] Calculated with the use of the CLOC-program found at http://cloc.sourceforge.net

5. Future Work

The following are examples of features that, given more time, could be implemented in our tool. A substantial amount of bugs still exist in our tool, some of them not yet uncovered due to the limited test coverage. Fixing the remaining bugs is a large potential area for future work. Support for simplification of logical expression could be added, such as: $boolExp == true \rightarrow boolExp$, $!!boolExp \rightarrow boolExp$ and more complex simplifications using DeMorganś law $((!boolExp1)||boolExp2) \quad \&\& \; boolExp1 \rightarrow boolExp1 \; \&\& \; boolExp2$

In the case of the following example:

```
import java.util.*;
import java.awt.*;
....
        List<Object> list;
```

Code example 14. On demand import conflict.

a conflict arises. Should List be imported from $java.awt$ or $java.util$? A feature which resolves such a conflict could be added to the tool, though it is possible to argue that this is handled already by the compilation error the conflict produces.

6. Discussion & Conclusion

The main problem of evaluating our tool was the incompleteness of the implementation. With strict time constraints it was impossible to implement functionality for all possible cases. This caused the tool to fail when used on the second test program; instead of cleaning the source code, the program is broken due to the removal of several used imports. When it does work, however, it accomplishes the goals that where set at the beginning of the project. With the test case of the snake game, for example, everything worked as intended. Further more, all the test cases written by ourselves passed. The problem with using test cases you write yourself is that it is very hard to achieve a high degree of test coverage[8] [9]. This lack of a total test coverage which surfaced only when evaluating our tool on a larger project is something which we would have liked to fix, unfortunately extending the functionality of the tool to include a larger array of Java functionality ended up outside the scope of the project.

The automatization of code cleanup brings with it some consequences which might not always be desirable. Functionality related to import on demand replacement might be undesirable due to the possibility that the user desire to continue to work on the modified source code without realizing that any new import has to be made directly. A better way to approach this feature would have been to make it optional, for example via a argument to the tool but due to time constraints this was never implemented. Looking at related work, Eclipse has taken a similar approach by separating import organizing and code cleanup.

Dr Ast was a great help when working with extensions of ExtendJ. Dr.Ast is a very useful graphical representation of the abstract syntax tree for a given file, enabling the user to see the relationships between different components of the program and their respective attributes. Another helpful component that simplifies the extending is the example files that comes with the compiler, providing the basis of the extension.

Even though usability is quite subjective, it is still useful to say something about it. A disadvantage of our tool is the current lack of a graphic interface. The option of using a GUI instead of text commands could increase the usability for many users. Something which is more arguable however, is the need for such a usability in

[8] P. Jalote, A Concise Introduction to Software Engineering, chapter 8

[9] J. Link, Unit Testing in Java: How Tests Drive the Code, chapter 8

a tool with a potential user base almost certainly well versed in the terminal. Even with the lack of a GUI the tool is not hard to use, it is a simple matter of executing the jar file and providing it with a java-file or several files as arguments.

References

Pankaj Jalote. A Concise Introduction to Software Engineering. *Springer-Verlag London Limited*, 2008.

Kent Beck. Extreme Programming Explained. *Addison-Wesley Boston, Massachusetts*, 2005.

Kiczales Gregor, Lamping, et al. Aspect-oriented programming in ECOOP'97 — Object-Oriented Programming: 11th European Conference. *pringer Berlin Heidelberg*, 1997.

Johannes Link. Unit Testing in Java: How Tests Drive the Code. *Morgan Kaufmann, Burlington Massachusetts*, 2003.

Torbjörn Ekman & Görel Hedin. The JastAdd Extensible Java Compiler. *OOPSLA 2007: 1-18, Proceedings of the 22nd Annual ACM SIGPLAN Conference on Objects-Oriented Programming Systems, Languages and Applications, Montreal Canada*, 2007.