Clone Removal in Java Programs as a Process of Stepwise Unification

Daniel Speicher and Andri Bremm

University of Bonn, Computer Science III,
Römerstraße 164, 53117 Bonn, Germany
dsp@cs.uni-bonn.de, bremm@uni-bonn.de

Abstract. Cloned code is one of the most important obstacles against consistent software maintenance and evolution. Although today's clone detection tools find a variety of clones, they do not offer any advice how to remove such clones. We explain the problems involved in finding a sequence of changes for clone removal and suggest to view this problem as a process of stepwise unification of the clone instances. Consequently the problem can be solved by backtracking over the possible unification steps.

Keywords: unification, backtracking, clone analysis, refactoring, program dependence graph, lambda expression

1 Introduction

In the last decades, practicable and reliable code clone detection tools [9] have been developed. These tools are able to find different kinds of clones, which are all important in some situations. Developers, who are interested in the removal of clones by refactoring [2], want to know whether and how a clone can be eliminated. A refactoring is a change to the source code that alters (typically improves) the design, but does not change the observable behavior of the software. We present an approach that gives precise refactoring suggestions depending on the set of available refactorings.

Fowler et al. provided in [2] a comprehensive catalog of such refactorings. A developer may use for example the RENAME METHOD refactoring to change the name of a method to a more expressive one. A tool that offers this refactoring has to make sure that the name of the method is not only changed in the declaration of the method, but as well at every method invocation. As the same name might be used for different methods in different scopes, a textual "search and replace" is not guaranteed to keep the observable behavior intact. The tool needs to know the Abstract Syntax Tree of the code including resolved bindings to methods. JTransformer [12] provides this information for Java programs as facts on which logic programs can reason.

Identical clones can be removed by EXTRACT METHOD, EXTRACT CLASS, PULL UP METHOD refactorings together with the appropriate adaptations at the

```
 1:   List<Person> adults() {
 2:         List<Person> adults = new ArrayList<Person>();
 3:         System.out.println("Retrieving adults");
 4:         for (Person p : persons) {
 5:               boolean isAdult = p.getAge() >= ADULT_AGE;
 6:               if (isAdult)
 7:                     adults.add(p);
 8:         }
 9:         return adults;
10:   }

11:   List<Person> children() {
12:         List<Person> children = new ArrayList<Person>();
13:         for (Person p : persons) {
14:               boolean isChild = p.getAge() < YOUTH_AGE;
15:               if (isChild)
16:                     children.add(p);
17:         }
18:         return children;
19:   }
```

Fig. 1. Two cloned methods. The methods differ in the name of two variables (`adults` vs. `children`, `isAdult` vs. `isChild`), one extra statement (line 3), and a non-unifiable expression (line 5 vs. line 14).

call side. If the clones have some *differences*, we suggest to start with the Program Dependence Graphs [3] (PDG) for the clone instances, to identify the statements that are *equal or unifiable* (i.e. can be made equal through refactorings) and finally rearrange the control flow based on the PDG so that all non-unifiable statements are separated. The example in the Figures 1 to 3 illustrates our approach.

2 Program Dependence Graphs

We start with two potential clone candidates. These may have been found with one of the existing clone detection tools like Simian [14] or Scorpio [6, 13]. For each of the clone candidates we build the PDG.

Such a PDG consists of one node for every statement and of two kind of edges representing control and data dependencies. There is a *control dependency* from a control statement to all directly enclosed statements. In our example the for-loop in line 4 controls the execution of line 5 and 6 while the execution of line 7 in turn is controlled by line 6. There is a *data dependency* from a statement s_1 to a statement s_2, if s_1 writes a variable that is read in s_2 and there is at least one possible execution on which s_1 is the last statement writing this variable before reaching s_2. In our example there is a data dependency from statement 2 to statement 9 as the for-loop might not be executed.

In extension to the established definition our data dependencies take as well method invocations into account. If a method returns a value without performing side effects we consider the method invocation only as a *read access* to the object. If the method does have side effects, we consider the invocation as a *write and read access* to the object. The PDG in Figure 2 gives an example of

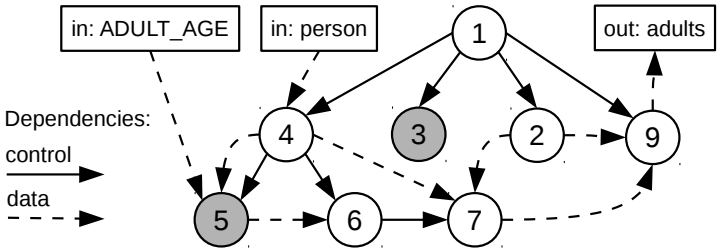

Fig. 2. A PDG of the method `adults` in Figure 1. The invocation of `add` in line 7 is interpreted as a write on `adults` leading to the data dependency from 7 to 9. The invocation of `getAge` in line 5 is interpreted as a read on `p`, so that there is no data dependency from 5 to 7. The extra statement in line 3 has no data dependencies, so that it can freely be moved below node 1. The data dependencies of the non-unifiable statement in line 5 forbid reordering but allow extracting into a lambda expression.

a data dependency resulting from this approach. This is still a heuristic way to transfer the concept of a data dependency to the object-oriented setting. Deeper analysis could label the data dependencies with a more precise characterization of the state that is changed by the method. In addition alias analysis could find additional hidden dependencies as a change to one object via one variable is a change to the object behind its aliases.

Once we have build the PDGs of the clone candidates, they are compared and nodes for equal or unifiable statements are mapped to each other. Whether two statements are unifiable depends on the refactorings that are considered.

3 Statement Unification

The RENAME refactoring allows us to consistently change the names of local variables and parameters or even fields and methods. Therefore we take at least the RENAME refactoring into account. Our example in Figure 1 and 3 illustrates this. If we consider further refactorings more statements become unifiable although some at the price of complexer parameter lists.

Differences in literals can be removed with the INTRODUCE PARAMETER refactoring. The GENERALIZE TYPES refactoring [7] allows to find differences in type declarations that are more specific than required by the usage of the declared object. If a type generalization is not possible e.g. because different specific return types are required by the callers of a method, the refactoring INTRODUCE TYPE PARAMETER can help. Finally method signatures that differ just in the order of parameters can be unified with the REORDER PARAMETERS refactoring.

In our example the difference between line 5 and line 14 can not be removed and the statement in line 3 has no counterpart in the second method. These differences require changes to the control flow.

4 Control Flow Unification

The PDG contains only as much information about the control flow as is relevant for the state of the variables at each line of the method. Therefore a node can freely be reordered (directly) below the node that controls it as long as the order of nodes with data dependencies is preserved.[1] This allows us to separate non-unifiable statement from unifiable statements, as it is the case with node 3 in our example.

Another possibility to "remove" non-unifiable statements is to use the EXTRACT METHOD refactoring. A group of contiguous statements is extractable if the corresponding nodes have to other statements only outgoing data dependencies for at most one variable and no outgoing control dependency [8]. In the PDG in Figure 2 the nodes 5, 6, and 7 together as well as the node 5 on its own is extractable.

The EXTRACT METHOD refactoring is especially helpful if the clones are in classes that are siblings in the class hierarchy. If in this case all differences can be extracted the remaining method can be PULLED UP to a common ancestor of the siblings. This sequence of refactorings is called FORM TEMPLATE METHOD and is explained in detail in [2].

If the classes containing the clones are unrelated the STRATEGY design pattern in combination with TEMPLATE METHOD may be used [1]. But, if there is only one or two differences and these differences are small, these pattern do not pull their weight and the introduction of lambda expressions is the method of choice.[2] The preconditions for the EXTRACT LAMBDA EXPRESSION refactoring are the same as for EXTRACT METHOD. Our example illustrates the use of lambda expressions to extract the difference between line 5 and 14.

5 Related Work and Conclusion

CloneDifferentiator [10] analyses and visualises the differences between the PDG of clones. The refactorings EXTRACT METHOD, INTRODUCE PARAMETER and the use of Generics are suggested. ARIES [5] calculates metrics to decide whether a refactoring is appropriate. For example EXTRACT METHOD is only recommended when the fragment refers to only a few variables outside the fragment.

[1] Tsantalis and Chatzigeorgiou in [8] correctly emphasize that there is one additional criterion to be considered. Although a write access to a variable *following* a read access can not influence the value of the variable, and therefore is not represented by a data dependency, moving the write access upwards may create a data dependency and change the behavior. Adding these so called "anti-dependencies" to the PDG an preserving their order solves the problem. Our example does not show anti-dependencies.

[2] Lambda expressions are essential for every functional language and have been available for some object-oriented languages as well. Finally they will be introduced to Java in the next version. The planed syntax for lambda expression in Java is explained in JSR 335 [4].

```
List<Person> adults() {
      System.out.println("Retrieving adults");
      return filterPersons((Person p) -> p.getAge() >= ADULT_AGE);
}

List<Person> children() {
      return filterPersons((Person p) -> p.getAge() < YOUTH_AGE);
}

interface PersonFilter { boolean test(Person p); }

List<Person> filterPersons(PersonFilter f) {
      List<Person> filtered = new ArrayList<Person>();
      for (Person p : persons) {
            boolean passesFilter = f.test(p);
            if (passesFilter)
                  filtered.add(p);
      }
      return filtered;
}
```

Fig. 3. The methods after the clone refactoring. The differing variables were renamed. The extra statement in the first method has been separated from the contiguous block of unifiable statements. The non-unifiable expressions have been extracted into lambda expressions. Lambda expressions consist of a parameter list surrounded by round brackets. The statements of the lambda expression follow after the arrow. If there is only one statement it is possible to omit the return keyword.

We described a process that derives for related clones one (or more) ways to remove the clones, by applying a series of refactorings. The parameters of the refactorings can be precisely (although not necessarily uniquely) derived from the context so that a tool can present precise refactoring suggestions to the developer. Elements of the presented approach such as the generation of the PDG are implemented as part of Cultivate [11].

The approach to search for a sequence of refactoring steps by exploring the different possibilities to unify statements and control flow naturally arises from the problem. As we start with clone candidates found by existing tools, the amount of data to be processed is limited: We know already which methods to compare and do not have to compare all possible method pairs. In addition typically only a few statements in the methods are unifiable, so that the graph matching is not as expensive as in the general case.

References

1. M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe and K. Kontogiannis, *Advanced Clone-Analysis to Support Object-Oriented System Refactoring.* Proceedings of 7th WCRE, 2000.
2. M. Fowler, *Refactoring: Improving the Design of Existing Code.* Boston, USA, Addison-Wesley, 1999.
3. J. Ferrante, K. Ottenstein, and J. Warren, *The program dependence graph and its use in optimization*, ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319?349, 1987.

4. B. Goetz, *JSR 335: Lambda Expressions for the Java^{TM} Programming Language.* `http://jcp.org/en/jsr/summary?id=335`, 2011.
5. Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue, *ARIES: refactoring support tool for code clone.* ACM SIGSOFT Software Engineering Notes 30, 1–4, 2005.
6. Y. Higo and S. Kusumoto, *Code Clone Detection on Specialized PDGs with Heuristics,* 15th European Conference on Software Maintenance and Reengineering, pp. 75–84, Mar. 2011.
7. F. Tip, A. Kieżun and D. Bäumer, *Refactoring for generalization using type constraints.* 18th Conference on Object-oriented programing, systems, languages, and applications, 13–26, 2003.
8. N. Tsantalis and A. Chatzigeorgiou, *Identification of Extract Method Refactoring Opportunities.* IEEE Transactions on Software Engineering 35, 2009.
9. C. Roy, J. Cordy, and R. Koschke, *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach.* Sci. Comput. Program 74, 470–495, 2009.
10. Z. Xing, Y. Xue and S. Jarzabek, *CloneDifferentiator: Analyzing clones by differentiation.* 26th International Conference on Automated Software Engineering, 576 -579, 2011.
11. "Cultivate": `http://sewiki.iai.uni-bonn.de/cultivate`
12. "JTransformer": `http://sewiki.iai.uni-bonn.de/jtransformer`
13. "Scorpio": `http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e`
14. "Simian": `http://www.harukizaemon.com/simian/`