Partial Redesign of Java Software Systems Based on Clone Analysis

Magdalena Balazinska[1], Ettore Merlo[1], Michel Dagenais[1], Bruno Lagüe[2] and Kostas Kontogiannis[3]

[1]Department of Electrical and Computer Engineering, École Polytechnique de Montréal,
P.O. Box 6079, Downtown Station, Montreal, Quebec, H3C 3A7, Canada
http://www.casi.polymtl.ca
e-mail: magda@casi.polymtl.ca {ettore.merlo,michel.dagenais}@polymtl.ca

[2]Bell Canada, Quality Engineering and Research Group
1050 Beaver Hall, 2nd floor, Montreal, Quebec, H2Z 1S4, Canada
e-mail: bruno.lague@bell.ca

[3] Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
e-mail: kostas@amorgos.uwaterloo.ca

Abstract

Code duplication, plausibly caused by copying source code and slightly modifying it, is often observed in large systems.

Clone detection and documentation have been investigated by several researchers in the past years. Recently, research focus has shifted towards the investigation of software and process restructuring actions based on clone detection.

This paper presents a new redesign approach developed for Java software systems. The approach factorizes the common parts of cloned methods and parameterizes their differences using the strategy design pattern. The new entities created by such transformations are also decoupled from the original contexts of their use thus facilitating reuse and increasing maintainability.

The applicability and automation of the technique presented in the paper have been verified by partially redesigning JDK 1.1.5.

1 Introduction

Source code reuse in object-oriented systems is made possible through different mechanisms such as inheritance, shared libraries, object composition, etc. Some design approaches, namely the well-known design patterns [6] particularly facilitate reuse. Nevertheless programmers often need to reuse components which haven't been designed for this purpose. This happens often when software systems go through the expansion phase and new requirements have to be satisfied periodically [6].

When such a situation arises, ideally, the modules involved should be restructured and the component properly reused. Even better, the whole system could be reorganized, classes could be refactored into general components and their interfaces rationalized. Such a process is known as consolidation and allows a system to become more flexible and easier to expand [6]. Unfortunately, often the process used instead is "cut-and-paste", i.e. performing some sort of reuse by manual source code inlining. This other approach produces what we call cloned pieces of code, or clones which will undergo independent successive maintenance [9].

Previous research has studied the detection of clones and has investigated their use for widely varying purposes including program comprehension, documentation, quality evaluation or system and process restructuring. Several techniques have been investigated in the literature for the detection of clones in software systems. Some techniques are based on a full text view of the source code. Johnson [7] has developed a method for the identification of exact duplications of substrings in source code using fingerprints whereas Baker's tool, "Dup" [2], reports both identical sections of code and sections that differ only in the systematic substitution of one set of variable names and constants for the other.

Other approaches, such as those pursued by Mayrand et al. [10] and Kontogiannis et al. [8] focus on whole sequences of instructions (BEGIN-END blocks or functions) and allow the detection of similar blocks using metrics. Those metrics relate to aspects of sequences of instructions such as their layout, the expressions inside them, their control flow, the variables used, the variables defined, etc.

In [8], Kontogiannis et al. also detect clones using two other pattern matching techniques namely dynamic programming matching which finds the best alignment between two code fragments, and statistical matching between abstract code descriptions patterns and source code.

Yet another clone detection technique relies on the comparison of subtrees from the AST (Abstract syntax tree) of a system. Baxter et al. [4] have investigated this technique.

Several applications of clone detection have also been investigated, Johnson [7] visualizes redundant substrings to ease the task of comprehending large legacy systems. Mayrand et al. [10] as well as Lagüe et al. [9] document the cloning phenomenon for the purpose of evaluating the quality of software systems. Lagüe et al. [9] have also evaluated the benefits in terms of maintenance of the detection of cloned methods.

The purpose of our research is to investigate the use of clones as a basis for those reengineering actions which are useful to the maintenance of systems. In this paper, we present a new technique that allows to factor common parts of cloned methods while parameterizing their differences using the *strategy* design pattern. Process changes like "problem mining and "preventive control" introduced by Lagüe et al. [9] are fully supported by the approach.

Merging the common parts of cloned pieces of code has already been investigated by Baxter et al. in [4]. Their approach, based on macros, allows the elimination of redundancies and thus the reduction of the quantity of source code in a system. Although macros are applicable to all detected cloned, since the semantics of differences is ignored, their use presents several drawbacks. It is restricted to languages that support macros but more importantly, when lexical changes are introduced to the macro, a manual verification is necessary to ensure that the intended semantic change correctly propagates to all the contexts of use of the macro.

The approach presented in this paper takes into account the meaning of clones and of their differences allowing the redesign to remain meaningful in the context of its use. The extraction of semantic differences between clones and their translation in terms of programming language entities is based on a detailed matching algorithm presented in Section 2. The main concepts underlying the redesign process are presented and discussed in section 3. The details of the process can be found in section 4.

The process has been applied to a real Java system, JDK 1.1.5. The results obtained from the redesign are presented and discussed in Sections 5 and 6.

```
function match(c: Grid; v1,v2: Sequence) => (cost: Integer)
  for ( i ← 1 to size(v1) )
    for ( j ← 1 to size(v2))
      tempCost ← computeCost(v1[i],v2[j])
```

$$\text{c[i][j].cost} \leftarrow \min \begin{cases} c[i-1][j].cost + 1, \\ c[i][j-1].cost + 1, \\ c[i-1][j-1].cost + tempCost \end{cases}$$

$$\text{c[i,j].previous} \leftarrow \begin{cases} c[i-1][j], \\ c[i][j-1], \\ c[i-1][j-1] & \text{depending on} \\ & \text{the minimal cost} \end{cases}$$

```
  return c[size(v1)][size(v2)]
```

Figure 1. Core method of the matching algorithm.

2 Matching algorithm

In order to take advantage of cloned methods for system redesign, the differences existing between them must first be extracted and analyzed. The extraction is performed using the algorithm briefly presented in this section. For a detailed discussion of the algorithm please refer to [3].

The comparison algorithm used is based on Kontogiannis et al.'s Dynamic Pattern Matching algorithm [8] in which a fundamental change has been performed: rather than aligning syntactically structured entities like statements, the new algorithm aligns syntactically unstructured entities like tokens. Afterwards the obtained alignment is projected onto the corresponding AST representations of the input sequence and the pattern sequence to produce a syntactically structured alignment which is used to analyze differences among code fragments. Hence, the dynamic matching is performed on vectors corresponding to the sequences of tokens forming the code fragments compared. A grid is used to compute and hold the detailed results of the match.

The core of the algorithm which is defined in function *match* is presented in Figure 1. Function *match* iterates over all the elements of the grid and computes the distance for consecutive sequences using previously computed distances between shorter sequences as well as the cost of matching the current tokens. This latter cost is determined by *computeCost*.

Function *computeCost* compares two tokens by testing for equality of types and values. Two nodes match perfectly if they belong to the same type, except if they're literals or identifiers. Then they must also have the same value. The function returns 0 if the tokens are equal and can be matched. Otherwise, it returns 2 (the equivalent of the cost of removing one token and then adding the other instead).

The optimal match or distance between two vectors is defined as the minimal amount of tokens that have to be inserted or deleted to transform one vector into the other.

Once the optimal match has been obtained, the correspondence between the sequences of tokens and the entities of the programming language has to be made. To achieve the correspondence, the source code must first be represented in a higher level of abstraction.

A number of program representation schemes have been proposed in the relevant literature. These include frames [13], annotated data and control flow graphs [15], Abstract Syntax Trees [12], logic formulas on program dependencies [5] and, relation tuples based on a language domain model [11].

We have chosen as a program representation scheme, the program's annotated abstract syntax tree (AST). We believe that this scheme is most suitable because:

- it does not require any overhead to be computed as it is a direct product of the parsing process and,
- it can be easily analyzed to extract programming language entities corresponding to the differences found during the comparison,
- it will be easy to manipulate during the redesign phase of the process.
- it's a machine usable format.

The tree is created during the parsing phase, and is annotated in a post-processing phase where linking information is added. An example AST is illustrated in Figure 2

Once the source code has been represented in this higher level of abstraction, the tokens forming the differences are linked to the corresponding AST elements. When consecutive tokens form a single difference, the proper ancestor is found. From the AST, the corresponding programming language entities are determined.

The algorithm finally returns a sequence of subtrees of the AST that represent differences between two code fragments. Those subtrees will be directly manipulated during the redesign phase of the process.

3 Making reuse explicit and increasing maintainability

As previously mentioned, the goal of the redesign proposed in this paper is to increase the explicit reuse of components and the maintainability of the system. A more maintainable system will be, among others, easier to modify

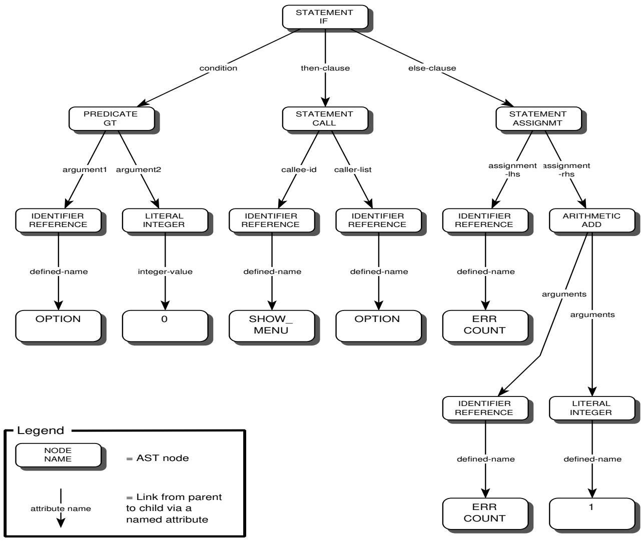

Figure 2. A sample AST.

and will present a smaller risk of unwanted side effects following modifications. To achieve the goal, the redesign process must factorize commonalities found in cloned methods and parameterize their differences to preserve the original behaviors of the clones. The newly created entity must also be decoupled from its environment to allow its further reuse and expansion.

In this section, a new structure which allows to produce such an entity is presented. It's based on the *strategy* design pattern.

3.1 The *strategy* design pattern

For an entity to partially change its behavior depending on use, some configuration mechanism must be provided. The challenge of producing a class containing a method with a partially configurable behavior can be met with the *strategy* design pattern. Indeed, as defined by Gamma et al. [6], "strategies provide a way to configure a class with one of many behaviors".

Figure 3 presents the *strategy* design pattern as described by Gamma et al. Three different kinds of classes participate in the pattern. The context class implements an operation containing some variable parts which can be the steps of an algorithm, accesses to complex data structures and so on. The context delegates the variable operations to the concrete strategy it's configured with. Hence, all the concrete strategies implement the same interface inherited from

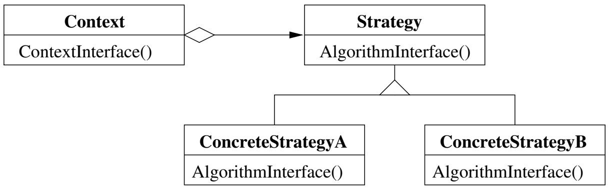

Figure 3. Structure of classes forming a *strategy* design pattern.

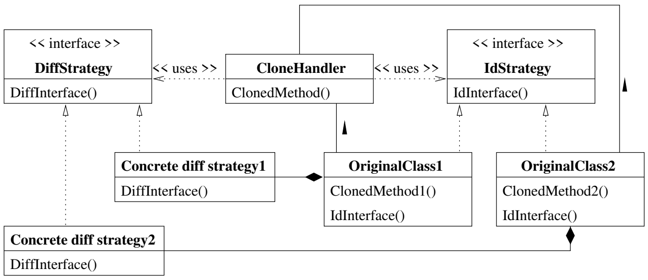

Figure 4. *Strategy* design pattern applied to clone removal.

the abstract strategy class. Variations in the implementation correspond to variations in behavior.

Both the context and the family of strategies are visible to clients. The latter must first choose a strategy and configure the context with it. Thereafter, they can interact exclusively with the context.

3.2 Application of the *strategy* design pattern to the parameterization of clone differences and their decoupling from context

Although cloned methods are often highly similar and accomplish almost the same operations, their output and side effects often differ in two aspects. First, different clones can manipulate different data structures and call different methods, even though those structures might have the same names. Indeed, except for parameters and local variables, the data structures manipulated and methods called depend on the class where each clone belongs, i.e. its context. Second, even similar clones are hardly ever identical. Usually, when a piece of code is copied and used in a different situation, it's somewhat modified: method calls, types of parameters or other elements are changed in its body. Therefore, part of the operations performed inside a cloned method varies from clone to clone.

Hence, when a new method in a new class is to replace some piece of code previously shared by different classes it has to be able to configure two aspects of its behavior. First, each time the method is called it has to be able to change the meaning of the identifiers present inside its body to match the meaning of those same identifiers in the context of the original clone, i.e. the context of the original clone class. Second, it has to be able to change those operations which are different in the different clones. On each call, it has to choose the right alternative for each difference. Those two requirements imply that the new class and method will have to be configured with two strategies: one allowing the decoupling from the original context and the other allowing the parameterization of the differences.

Figure 4 shows an original use of the *strategy* design pattern that allows the factorization of clone commonalities, the parameterization of their differences and the decoupling of the shared entity from the original contexts of its use. The new class containing the factored source code is represented by the CloneHandler class which uses two strategies for its configuration. CloneHandler contains the new ClonedMethod which will handle the cloned operation as a whole while delegating the variable parts to one of the two strategies. Necessary identifier redirections will be delegated to the *IdStrategy* whereas difference handling will be delegated to the *DiffStrategy*. Each one of the two strategies corresponds to a different interface.

The original classes represent the concrete *IdStrategies*. They will handle all the operations necessitating the access to their attributes or methods. Each of the original classes is also made to aggregate one or several new classes which represent the *DiffStrategies*. One *DiffStrategy* is created per cloned method. It will handle the operations unique to that clone.

An important particularity of the structure of classes obtained is that the original classes are also the clients of the new CloneHandler class. It makes the relationship cyclical: the original classes delegate the handling of the whole cloned operation to the new class which then delegates back context dependent operations.

Figure 4 also shows that instead of inheriting an abstract strategy as proposed by Gamma et al., the concrete strategies implement interfaces. Such a modification allows not to modify the existing relations between the original classes and avoids multiple inheritance when original classes already have parents.

Finally, it's important to note that cloned pieces of code don't have to correspond to complete methods. Indeed, if a piece of code of arbitrary length and position has to be moved to a new class, it can be encapsulated in a method and replaced by a method call. Of course, the parameters it needs will have to be determined but the task is quite trivial when using a proper symbol table.

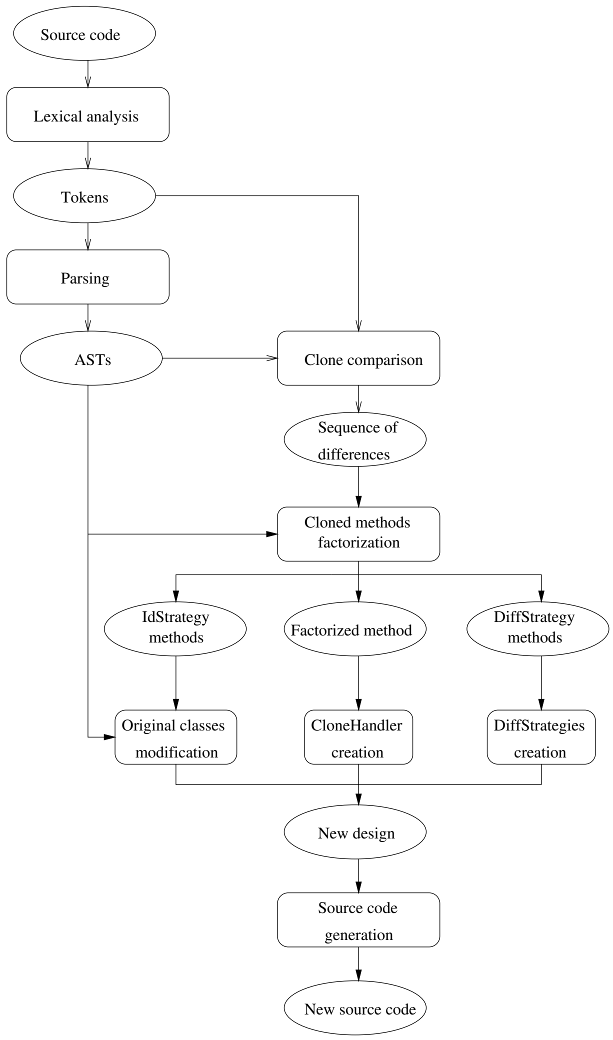

Figure 5. Redesign process.

4 Redesign process

To accomplish the transformations described in the previous section, a precise redesign process has been developed. Figure 5 shows the phases of the process which takes as input a group of cloned methods and outputs the new source files ready to be compiled.

The main phases of the process are the following:

- The comparison algorithm described in Section 2 is first used to determine the differences between the cloned methods *ClonedMethod1*, *ClonedMethod2*,... *ClonedMethodN* from the original classes *OriginalClass1*, *OriginalClass2*,... *OriginalClassN*.

- *ClonedMethod1*, *ClonedMethod2*,... *ClonedMethodN* are transformed into the single *CloneMethod* method which will become a member of the new *CloneHandler* class.

- The original classes and methods are modified using the following main steps:

  – For each cloned method of each class, an inner class is created and instantiated. It will represent the *DiffStrategy* of the cloned method.

  – The bodies of the cloned methods are replaced by method calls to the new method *ClonedMethod*. Those method calls will forward to *ClonedMethod* the requests originally handled by the clones.

  – Methods created during the phase of merging the cloned source codes are inserted into the original classes or the inner classes depending if they relate to redirections or difference handling.

- The interfaces *IdStrategy* and *DiffStrategy* are built by adding to them all the necessary signatures.

- The new *CloneHandler* class containing the new method *ClonedMethod* is created.

- Finally the new source files are generated.

Figure 6 presents two cloned methods, *setStartRule* and *setEndRule* extracted from the file SimpleTimeZone.java (package java.util) of JDK 1.1.5. Figure 7 presents the same file after reengineering. Figure 8 and 9 contain the source code of the new class and the new interfaces created.

The following subsections detail the different phases of the redesign process.

4.1 Factorizing cloned methods commonalities

To merge the source code of different clones, two kinds of actions have to be performed. First, all the differences found during the comparison of the methods have to be replaced by appropriate structures so their handling could be parameterized with the *DiffStrategy*.

Those transformations are facilitated by the fact the the comparison algorithm introduced in Section 2 abstracts the source code and the differences between clones into ASTs thus obtaining syntactically structured differences which are appropriate to parameterization and meaningful in terms of the programming language.

Second, all the remaining identifiers have to be found in the symbol table to determine whether they're visible in the new class and can be used directly or they're not visible and have to be manipulated through method calls (through the *IdStrategy*).

```
File: SimpleTimeZone.java
-----------------------------------------------------------
package java.util;
import java.io.ObjectInputStream;
import java.io.IOException;
  //----------------------------------------
public class SimpleTimeZone extends TimeZone {
    ...
  private int startMonth, startDay,
              startDayOfWeek, startTime;
  private int endMonth, endDay,
              endDayOfWeek, endTime;
  private boolean useDaylight = false;
    //----------------------------------------
  public void setStartRule(int month,
                           int dayOfWeekInMonth,
                           int dayOfWeek,int time)
    {
    startMonth = month;
    startDay = dayOfWeekInMonth;
    startDayOfWeek = dayOfWeek;
    startTime = time;
    useDaylight = true;
    }
    //----------------------------------------
  public void setEndRule(int month,
                         int dayOfWeekInMonth,
                         int dayOfWeek,int time)
    {
    endMonth = month;
    endDay = dayOfWeekInMonth;
    endDayOfWeek = dayOfWeek;
    endTime = time;
    useDaylight = true;
    }
  }
```

Figure 6. Cloned methods extracted from JDK 1.1.5.

4.1.1 Parameterizing differences

Most of the differences can be encapsulated into methods and replaced by method calls. For each of such differences, the kind of method appropriate for its encapsulation has to be determined, the method has then to be created and the difference has to be replaced by the corresponding method call.

The choice and use of the appropriate method to replace a difference can be determined using the following rules:

- If the difference is a method call, it can be replaced by a call to a unifying method from the *DiffStrategy* which will redirect the call to the appropriate method. For example, let's suppose that at some point, one clone uses method doSomething() and another uses method doSomethingElse() instead. In the factored code, the difference can be replaced with a call to stratDiffs.difference_i() with stratDiffs being the parameter referring to the appropriate strategy and methods difference_i() being defined as (let's suppose both methods returned an int value):

```
File: SimpleTimeZone.java
-----------------------------------------------------------
package java.util;
import java.io.ObjectInputStream;
import java.io.IOException;
  //----------------------------------------
public class SimpleTimeZone extends TimeZone
               implements newInterf.STZone_InterfID {
    ...
  private int startMonth, startDay,
              startDayOfWeek, startTime;
  private int endMonth, endDay,
              endDayOfWeek, endTime;
  private boolean useDaylight=false;
    //----------------------------------------
  public void setStartRule( int month,
                            int dayOfWeekInMonth,
                            int dayOfWeek,int time) {
    STZone_handler.setStartRule(month,dayOfWeekInMonth,
             dayOfWeek,time, STZone_setStart_strategy);
    }
    //----------------------------------------
  public void setEndRule( int month,
                          int dayOfWeekInMonth,
                          int dayOfWeek,int time) {
    STZone_handler.setStartRule(month,dayOfWeekInMonth,
              dayOfWeek, time, STZone_setEnd_strategy);
    }
    //----------------------------------------
  class STZone_setStart
            implements newInterf.STZone_InterfDiffs {
    public STZone_setStart() {
      }
    public void set_startMonth( int x) {
      startMonth = x;
      }
    public void set_startDay( int x) {
      startDay = x;
      }
    public void set_startDayOfWeek( int x) {
      startDayOfWeek = x;
      }
    public void set_startTime( int x) {
      startTime = x;
      }
    }
    //----------------------------------------
  class STZone_setEnd
            implements newInterf.STZone_InterfDiffs {
    public STZone_setEnd() {
      }
    public void set_startMonth( int x) {
      endMonth = x;
      }
    public void set_startDay( int x) {
      endDay = x;
      }
    public void set_startDayOfWeek( int x) {
      endDayOfWeek = x;
      }
    public void set_startTime( int x) {
      endTime = x;
      }
    }
    //----------------------------------------
  public void set_useDaylight( boolean x) {
    useDaylight = x;
    }
    //----------------------------------------
  private newHandlers.STZone_Handler STZone_handler
          = new newHandlers.STZone_Handler(this);
  private STZone_setStart STZone_setStart_strategy
          = new STZone_setStart();
  private STZone_setEnd STZone_setEnd_strategy
          = new STZone_setEnd();
  }
```

Figure 7. Cloned methods from Figure 6 after reengineering.

```
File: STZone_Handler.java
----------------------------------------------------------
1 package newHandlers;
2 import java.io.ObjectInputStream;
3 import java.io.IOException;
4 import java.util.*;
  //----------------------------------------
5 public class STZone_Handler {
6   private newInterf.STZone_InterfID _stratID;
    //----------------------------------------
7   public STZone_Handler(newInterf.STZone_InterfID
8                                            stratID) {
9     _stratID = stratID;
    }
    //----------------------------------------
10  public void setStartRule(int month,
11                           int dayOfWeekInMonth,
12                           int dayOfWeek, int time,
13               newInterf.STZone_InterfDiffs stratDiffs) {
14    stratDiffs.set_startMonth( month);
15    stratDiffs.set_startDay( dayOfWeekInMonth);
16    stratDiffs.set_startDayOfWeek( dayOfWeek);
17    stratDiffs.set_startTime( time);
18    _stratID.set_useDaylight( true);
    }
  }
```

Figure 8. New class obtained from merging the source code of the cloned methods from Figure 6.

```
...inside the first strategy:
public int difference_i() {
    return doSomething();
}

... inside the second strategy:
public int difference_i() {
    return doSomethingElse();
}
```

- It the difference is a non-local variable, the method that can replace the difference is a "get" or "set" method on the variable. The example of figures 6 through 9 shows such transformations. For example lines 13 and 21 from file SimpleTimeZone.java are factored by being replaced by the method call of line 14 in file STZone_Handler.java.

  The choice of the "get" or "set" methods depends on the context of use of the variable. The argument to the "set" method is also determined by the context. The type of the parameter or return value is determined from the symbol table entries of the variables.

- If the difference is a constant, the type of the constant must be determined and a "get" method must be created and used.

- If the difference is an expression, a statement or a block of statements, the subtree has to be carefully examined to determine the variables used and the expression returned. Then the subtree can be encapsulated in a method and replaced by a method call.

```
File: STZone_InterfID.java
----------------------------------------------------------
package newInterf;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.util.*;
  //----------------------------------------
public interface STZone_InterfID {
  public void set_useDaylight(boolean x);
   }
----------------------------------------------------------
File: STZone_InterfaceDiffs.java
----------------------------------------------------------
package newInterf;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.util.*;
  //----------------------------------------
public interface STZone_InterfDiffs {
  public void set_startMonth(int x);
  public void set_startDay(int x);
  public void set_startDayOfWeek(int x);
  public void set_startTime(int x);
  }
```

Figure 9. Interfaces created after reengineering clones from figure 6.

- If the difference is a list of arguments, each argument has to be enclosed in a method call separately.

When the differences reside in types of parameters, local variables or the return type, the treatment of the difference is more complex but the following rules can be used:

- If the difference is the return type, the method should be made to return an Object type. The appropriate type cast must be used when recuperating (in the original methods) the object returned.

- If the difference is a parameter type, an appropriate parent has to be identified. If the interface of the parent corresponds to what is used inside the method, no other manipulation has to be performed. If such is not the case, each manipulation of one of the parameters has to be encapsulated in a method and delegated to the strategy.

- If the difference resides in the type of a local variable, one of the creational design patterns such as factory has to be used to defer to the *DiffStrategy* class the choice of the type of the entity to create.

4.1.2 Decoupling the common parts from their original contexts

Once the differences have been taken care of, all the remaining identifiers found in the body of the clones have to be analyzed. Indeed, all the methods, attributes and constants declared in the original classes, their ancestors or their enclosing classes are context dependent elements from which the new method must be properly decoupled. Hence, their use has to be changed as follows:

- Method calls are changed so that they are accessed through the reference to the *IdStrategy* as in stratID.doSomething(). The only problem is when the methods are private or protected. Currently, the problem is solved by changing the visibility of the methods to public.
- The attributes and constants are treated as differences except for the strategy used to manipulate them. An example of such a transformation is the factorization of lines 17 and 25 from file SimpleTimeZone.java (figure 6) into line 18 from file STZone_Handler.java. (figure 8).

The other identifiers aren't context dependent and require only simple manipulations. Local variables and parameters are left as they are because they are still visible when the code is moved between classes. Methods, types and constants which are declared in classes or interfaces unrelated to the original classes might not be directly accessible in the new class. Their visibility depends on the imports and package declarations. To ensure that their visibility will be the same in the new class as it was in the original classes (and so to avoid redirecting their accesses), all the imports made in the files containing the original classes are copied to the files containing the new class and interfaces. All the elements in the packages to which the original classes belong are also imported in the new class and interface.

4.2 Transforming the original classes

Once the bodies of the cloned methods have been factored, they are replaced in the original classes by calls to the new method. The method call is made through a reference to an instance of the new class *CloneHandler*. This reference is declared as a private attribute of each of the original classes. Figure 7 shows an example of such a transformation on lines 14-15 and 19-20. When the call is made, an extra argument is added to the called method. This argument is the concrete *DiffStrategy* that is passed to the new method so the latter will be able to configure its behavior.

All the original classes are also made to implement the new interface *IdStrategy* because they represent the concrete *IdStrategies*. Methods created during the phase of merging the cloned source codes and treating the identifiers are inserted into the original classes.

For each cloned method present in a class, an inner class and a corresponding private attribute are created and inserted into the original classes. Each inner class will represent the *DiffStrategy* of one cloned method. They will contain all the methods needed to handle differences between the corresponding cloned methods and other cloned methods. Such difference handling methods were created during the factorization of the source codes and more precisely the phase of difference treatment. Each inner class is also made to implement the new interface *DiffStrategy* which represents the behavior configuration strategy.

4.3 Building the strategy interfaces

The strategy interfaces are simply built by including in them all the necessary methods signatures. All the imports made in the original classes are also inserted into the interface as are all the elements of the packages to which the original classes belong. The new interfaces are declared as belonging to a new package. All the new interfaces belong to the same package.

4.4 Creating the new class

The new class is created by setting all the imports as they have been set in the new interfaces and declaring the class to belong to the package of clone handlers.

A reference to an *IdStrategy* class is declared along with a constructor that sets it with a value received in parameter. Indeed, the *IdStrategy* doesn't change over the life span of an instance of the class.

Finally, the factored code of the cloned methods is inserted into the new class.

5 Applying the redesign process

The process presented in the previous section has been implemented in CloRT (Clone Reengineering Tool). The tool has been developed in Java, using JDK 1.1.7. To get the ASTs of the source files, CloRT uses a java parser generated with Javacc version 8 (first pre-release).

We have applied CloRT to JDK 1.1.5 [14], a development kit from Sun Microsystems with 145 000 lines of code. The redesign was conducted on a Pentium Pro 180MHz with 64MB RAM running Linux version 2.0.27.

CloRT implements the reengineering of identical clones, clones presenting superficial differences (names of local variables, names of parameters and name of the method), and clones differing in the use of non-local variables.

Table 1. Impact of the reengineering of 28 clones extracted from JDK 1.1.5

	Identical Clones	Global variables
number of clusters reengineered	6	5
source code variation in LOCs	+512	+545
percent variation of the total size of the system	+0.7	+0.8
files affected	13	7
reengineered classes	13	7
reengineered methods	14	14
classes created	6	5
interfaces created	12	10
methods created	44	40

Therefore clones corresponding to the three mentioned categories have been isolated from the other clones extracted from JDK. To isolate the three categories of clones, all the clones have been categorized using SMC (Similar Method Classifier) [3]. SMC classifies similar methods into 18 categories based on the meaning of the differences existing between clones. Finally, 28 cloned methods grouped in 11 clusters have been used for the redesign process.

After redesign, the corresponding source code has been automatically synthesized. The obtained system has been successfully compiled to create a new version of JDK. Simple programs which exercise some of the modifications have been successfully executed in the redesigned and regenerated environment.

6 Discussion

6.1 Impact on the maintenance of the system

The application of the redesign process to real systems allows to increase maintainability and explicit reuse. The redesign is in agreement with Lagüe et al.'s [9] "problem mining" process that aims at coping with the existing base of software clones in a system by allowing programmers to see the clones of any piece of code they're changing and determine if the changes should be propagated to the other copies of the code or not. Indeed, our redesign process allows changes affecting the common part of clones to propagate automatically to all copies whereas changes affecting unique characteristics to remain local.

Moreover, because the new structure clearly separates shared source code from differences, changes made to any method of a group of related methods won't propagate to any other method of the group. The modifications simply have to be introduced in the appropriate *DiffStrategy* class.

Even if new differences have to be introduced between related methods, the transformation remains of little complexity. The differences need only to be encapsulated in methods and used through method calls.

Similarly, if the common part of related methods is to be changed, the change applies automatically to all the methods. The detection of the classes affected by the impact is also straightforward. Thus the modification of existing features of a system should be simple after the system has been reengineered.

Keeping track of transformed clones and files added to the system is also of little difficulty as the interfaces and classes created are grouped into separate packages.

Another advantage of the newly created structures is their extensibility. If the source code of one of the shared method needs to be reused again in some other class or module, a proper and easy reuse is possible. No matter where the method is needed, the only manipulations that have to be performed are the addition of a reference to the class containing the factored code and the creation of the methods and inner classes that will handle the differences and attribute manipulations. The latter are very simple methods, so their creation is of little difficulty. Therefore, not only is the reuse simple but the overhead is small.

6.2 Impact on program comprehension

From Figures 6 through 9, it appears that the new source code is easy to understand. The *strategy* design pattern allows to parameterize the parts that depend on the original methods or on their classes. The methods created are also quite simple and short which eases their comprehension.

6.3 Impact on the size of the system

Table 1 shows the impact of the redesign in terms of variation in the size of the source code, quantity of classes, interfaces and methods. One notices that the reengineering activity actually increases the size of the source code as it happens in many reengineered systems. This result is quite interesting. In fact, merging cloned methods decreases the total size of source code but when this code is moved into a new class, short methods to access and modify internal attributes of the original classes have to be created. Moreover, methods that correspond to differences between clones have also to be added to the new inner classes. As the table shows as much as 84 methods have been created while merging the source code of 28.

Actually, this result is in agreement with the trend of well designed object-oriented systems which present more but shorter methods than more tightly coupled systems.

As the example presented in Figures 6 through 9 shows, the overhead of the new structures is quite important. Thus, the variation of the size of source code is smaller when methods cloned are longer, contain proportionally less differences and more methods form each group of clones. The overhead could be reduced by making CloRT verify the existence of methods allowing manipulations of the attributes of classes before creating such methods. Currently, CloRT only ensures that it never creates twice the same method even in a different execution. CloRT also creates only the methods that are really necessary. If, for example, an attribute is used without being modified only a "get_attribute()" method will be created without the corresponding "set_attribute(...)".

6.4 Difficulties in the application of the process

The major difficulty related to the process is the degree of complexity of some manipulations such as the determination of the return type of a complex expression.

Also, synchronized methods are currently not considered by the process. Indeed, the lock associated with the cloned methods must be moved from the original classes to the new class. Static clones aren't transformed either because they would require a different implementation of the strategy that handles the identifier redirections.

Finally, the overhead associated in the new method, with the redirections of attributes through a method call and an interface may have an impact on the execution time of the method. Nevertheless, such a difference shouldn't bear any consequences in most cases. Critical methods from the point of view of the execution time can simply be omitted during redesign.

7 Conclusions and future work

This paper has presented a new automatic redesign process that allows to transform the implicit reuse of source code by cloning into an explicit reuse of an independent component. The transformation is accomplished through the factorization of commonalities found between clones, the parameterization of their differences and the decoupling of the newly created entity from its original contexts of use.

The process has successfully been applied to three categories of clones found in JDK 1.1.5. Duplicated pieces of code have been replaced with more appropriate structures that allow explicit reuse and promote a high degree of extensibility and thus facilitate maintenance. The new structures created are also easy to understand and manage.

The results of the reengineering activities conducted show an increase in the total size of source code due to the numerous but simple methods created.

Finally, the experiment shows that it's possible to replace clones with more maintainable structures totally automatically, i.e. without any input from users nor any information on the system besides its source code.

The next step in the analysis of redesign based on clone information is to investigate the redesigns of other categories of clones [3] and to experiment with other design patterns and structures possible in Java or in other programming languages.

8 Acknowledgements

This research project has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Bell Canada.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-wesley, 1988.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*. IEEE Computer Society Press, July 1995.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *International Symposium on Software metrics. METRICS'99*. IEEE Computer Society Press, November 1999.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntaxt trees. In *Proceedings of the International Conference on Software Maintenance 1998*, pages 368–377. IEEE Computer Society Press, 1998.

[5] G. Canfora, A. Cimitile, and A. DeLucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance 1994*, pages 424–433. IEEE Computer Society Press, 1994.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-wesley, 1997.

[7] J. H. Johnson. Identifying redundancy in source code using fingerprints. *CASCON'93*, pages 171–183, October 1993.

[8] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:77–108, March 1996.

[9] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance 1997*, pages 314–321. IEEE Computer Society Press, 1997.

[10] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance 1996*, pages 244–253. IEEE Computer Society Press, 1996.

[11] H. Müller. Understanding software systems using reverse engineering technology perspectives from the Rigi Project. *CASCON'93*, pages 217–226, October 1993.
[12] P. Newcomb and P. Scott. Requirements for advanced year 2000 maintenance tools. *IEEE Computer*, pages 52–57, March 1997.
[13] J. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, 1994.
[14] Sun Microsystems Inc. Jdk 1.1.5.: Java development kit.
[15] L. Wills. Automated program recognition by graph parsing. MIT Technical Report 1358, MIT, AI Laboratory, 1993.