Advanced clone-analysis to support object-oriented system refactoring

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
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

[3] Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

ABSTRACT

Manual source code copy and modification is often used by programmers as an easy means for functionality reuse. Nevertheless, such practice produces duplicated pieces of code or clones whose consistent maintenance might be difficult to achieve. It also creates implicit links between classes sharing a functionality. Clones are therefore good candidates for system redesign.

This paper presents a novel approach for computer-aided clone-based object-oriented system refactoring. The approach is based on an advanced clone analysis which focuses on the extraction of clone differences and their interpretation in terms of programming language entities. It also focuses on the study of contextual dependencies of cloned methods. The clone analysis has been applied to JDK 1.1.5, a large scale system of 150 KLOC.

Keywords

Clone analysis, refactoring, redesign, maintenance

1 Introduction

Source code reuse in object-oriented systems is made possible through different mechanisms such as inheritance, shared libraries, object composition, and so on. Nevertheless programmers often need to reuse components which haven't been designed for this purpose. This may happen when software systems go through the expansion phase and new requirements have to be periodically satisfied [8].

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 [8]. Often, the process used instead is a manual "copy-and-paste". This other approach produces what we call cloned pieces of code, or clones which will undergo independent successive maintenance [12].

The goal of our research is to investigate the use of clone information as a basis for object-oriented system refactoring. Clones are good candidates for redesign as they represent duplicated code whose consistent maintenance might be difficult to achieve. They also form implicit links between components that share a functionality. Detection of clones in large software systems has been investigated in the past by [2, 7, 9, 10, 13] while clone elimination or reduction has been investigated by [3, 4, 5].

In [3] we have investigated the problem of classifying clones according to their types and according to the opportunities of further reengineering actions. In [4] an automatic approach which allows the factorization of common parts of clones, hence removing duplications while preserving the unique behavior of each clone, has been proposed. Such an approach was limited to those reengineering actions which ended up in a "strategy-based" design pattern.

The approaches presented in [3, 4] could be advantageously enhanced by targeted user interactions. Such interactions could alleviate the complexity of analysis necessary in an automatic approach and would give more flexibility to the user. Furthermore additional "design patterns" (more than "strategy" only) and reengineering actions should be investigated.

In this paper, we therefore propose a new approach oriented towards computer-assisted clone refactoring. The main goal is to significantly support refactoring decisions by providing detailed and relevant information on clones but letting the programmer decide on the actual refactoring to be performed, i.e., on the choice of candidates for refactoring, on the appropriate type of redesign and on the actual redesign actions. Those actions could then be performed with specialized tools such as the Refactoring Browser [6]. Such a computer-aided refactoring must be based on a powerful clone analysis. Also, a new redesign scheme based on "template" design-pattern is presented and discussed.

Most clone analysis approaches [2, 7, 9, 12] provide the user only with information on the amount, location and size of clones in a system. Some approaches [5, 10, 13] also provide a certain information on the degree of similarity of clones. Those information are nevertheless insufficient to support refactoring activities as they don't provide any insight on possible refactoring actions. To use clones as a basis for object-oriented system redesign, programmers need to know the exact differences between cloned methods and those differences must be meaningful in terms of corresponding programming language entities. Indeed, differences determine refactoring actions. Programmers also need to know the coupling between cloned methods and their contexts of use as such coupling will determine the overhead of transferring code fragments between classes.

This paper proposes an innovative and more advanced clone analysis which can be helpful in object-oriented system refactoring. This analysis determines detailed information on differences between cloned methods and their contextual dependencies. Sections 2 and 3 present those two aspects of the advance clone analysis; Section 2 presents the analysis of differences between clones, whereas Section 3 details the analysis of context dependent operations found in cloned methods. Section 4 presents the results of the application of those analyses to JDK 1.1.5. Finally, Section 5 discusses the use of the information provided by the advance clone analysis for object-oriented system refactoring. Section 6 presents related work on clones.

2 Difference analysis

During clone based refactoring, programmers need to know the differences between clones, and those differences must be available to them in a readily understandable format. This knowledge will determine the exact refactoring actions that will have to be performed.

This section describes the difference analysis aspect of the novel clone analysis. The algorithm that allows the extraction of differences is first described, followed by the approach that allows the interpretation of those differences as programming language entities. The generalization of the approach to more than two code fragments is then presented. Finally, the analysis of kinds of differences and the computation of more synthesized difference information are presented and their use for refactoring is emphasized.

Matching code fragments

The extraction of clone differences 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 [11] in which a fundamental change has been performed: rather

$$
\begin{array}{l}
\text{function match(c: Grid; v1,v2: Sequence) => (cost: Integer)}\\
\quad \text{for (i} \leftarrow \text{1 to size(v1))}\\
\quad\quad \text{for (j} \leftarrow \text{1 to size(v2))}\\
\quad\quad\quad \text{tempCost} \leftarrow \text{computeCost(v1[i],v2[j])}\\
\quad\quad\quad \text{c[i][j].cost} \leftarrow \min \begin{cases} c[i-1][j].cost+1, \\ c[i][j-1].cost+1, \\ c[i-1][j-1].cost+tempCost \end{cases}\\
\quad\quad\quad \text{c[i,j].previous} \leftarrow \begin{cases} c[i-1][j], \\ c[i][j-1], \\ c[i-1][j-1] & \text{depending on} \\ & \text{the minimal cost} \end{cases}\\
\quad \text{return c[size(v1)][size(v2)]}
\end{array}
$$

Figure 1: Core method of the matching algorithm.

than aligning syntactically structured entities like statements as is often suggested in the literature [5, 11], the new algorithm aligns syntactically unstructured entities like tokens. The comparison is hence performed at the lexical level with a reasonable complexity of $\theta(n*m)$ (where n and m are the sizes of the code fragments) that could even be decreased with a beam search optimization. Moreover, the algorithm provides a very fine grained match. The optimal match or distance between two code fragments is defined as the minimal amount of tokens that have to be inserted or deleted to transform one code fragment into the other.

The dynamic matching is hence performed on vectors corresponding to sequences of tokens forming the code fragments compared. A cost 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 result is then represented as a set of sequences of tokens that correspond, or that have to be inserted or deleted (consecutive insertions and deletions correspond to substitutions):

$$Match = < (s_1v_1, s_1v_2, action_1), (s_2v_1, s_2v_2, action_2), \\ ..., (s_kv_1, s_kv_2, action_k) > \quad (1)$$

Where, $\forall i \in [1..k]$:

- $s_iv_1 \subseteq v1$ and $s_1v_1 \frown s_2v_1 \frown ... \frown s_kv_1 = v1$ where $\frown$ is the concatenation operator.
- $s_iv_2 \subseteq v2$ and $s_1v_2 \frown s_2v_2 \frown ... \frown s_kv_2 = v2$
- $action_i \in \{match, addition, deletion, substitution\}$

Projecting differences
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. It will provide programmers with information at the appropriate level of abstraction to serve as a basis for refactoring. To achieve the correspondence, the source code must first be represented in a higher level of abstraction. We have chosen the program's annotated abstract syntax tree (AST) as a program representation scheme as it can, among others, be easily analyzed to extract programming language entities corresponding to differences found during the comparison.

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. Each token corresponds to exactly one node in the AST and is therefore linked to that node. When consecutive tokens belong to a single difference, the first ancestor of the nodes corresponding to those tokens is found.

This approach of first aligning tokens and then projecting them onto the AST rather than directly comparing ASTs, as is often suggested in the literature [5, 11] allows to get a very detailed match with a low computation complexity.

From the AST, the corresponding programming language entities are determined. The set of differences is finally obtained as:

$$Differences = \mathcal{P}(Trees_1 * Trees_2) \quad (2)$$

Where $\mathcal{P}(s)$ denotes the power set of s whereas $Trees_1$ and $Trees_2$ are the sets of all subtrees of the ASTs of the code fragments.

Comparing more than two code fragments
The approach described above gives the set of differences when exactly two code fragments are compared. Often, clusters of clones contain more than two cloned methods. To determine the exact set of differences between more than two code fragments, the following approach is taken.

- Let $Clones = \{C_1, C_2, ..., C_n\}$ be the set of clones in the cluster.
- C_1 is arbitrarily chosen as the reference code fragment.
- The optimal matches $Match(C_1, C_i)$ are computed for all the values of i in [2..n].
- For each $Match(C_1, C_i)$, the sequences s_jv1 of the reference vector for which $action_j \neq match$, i.e., sequences corresponding to differences (insertions, deletions or substitutions) are propagated to all the other matches defined by: $\{Match(C_1, C_m) \mid \forall m \in [2..n] \; with \; m \neq i\}$.
- The propagation of differences may produce adjacent sequences corresponding to differences. Such sequences are merged.
- After the propagation of the differences, sequences s_jv1 of the reference vector become identical in all $Match(C_1, C_i)$. The union of all differences is then straightforward.
- The differences found in each vector of tokens are then projected onto the corresponding ASTs.

The set of differences is finally obtained as:

$$Diff = \mathcal{P}(Trees_1 * Trees_2 * ... * Trees_n) \quad (3)$$

Where $Trees_i$ denotes the set of all subtrees of the AST of C_i. In the following sections, for each difference $d : Diff$, $d[i]$ will refer to the i th. subtree involved in that difference.

Discriminating kinds of differences
Once code fragments have been matched and their differences have been projected onto the AST, a very detailed and useful knowledge of clones has been gained. Textual differences between code fragments have been interpreted as programming language entities meaningful to the programmer.

Providing the programmer with the set of differences and their exact meaning is an already very useful information, especially in latter phases of refactoring when specific reengineering actions are being performed. The difference analysis moves the interpretation one step further, though. It provides difference information in a more synthesized manner, helpful in less advanced phases of refactoring. The differences are grouped, based on their role in refactoring.

From previous research on automatic refactoring [3, 4], we have determined that all differences don't affect refactoring in the same manner. More precisely, the distinguished between the following kinds of differences is useful:

```
function groupDiffs()
   RV = N = M = TE = TypeDiffs = Other = ∅
   ∀ d ∈ Diff
      Type = choseSet(d)
      Type = Type ∪ {d}
   Other = Diff \ ( RV ∪ N ∪ M ∪ TE ∪ TypeDiffs )
```

Figure 2: Difference information synthesis algorithm.

- Superficial differences such as names of parameters or names of local variables don't affect the behavior of methods nor their outputs. They're therefore discarded at this phase of the analysis.

- Differences affecting the signature of methods: return value, modifiers (static, public, and so on), names or list of thrown exceptions have to be carefully treated during redesign even though they don't directly affect common code fragments.

- Differences affecting the types of parameters or local variables or types are explicitly manipulated in *typecasts* or *instanceof* expressions make the transformation of clones into a general component more complex, especially in languages such as Java, that don't allow parameterizable types.

- All the other differences.

The discrimination between kinds of differences gives a more synthesized perspective as it groups differences along their role in refactoring. Programmers are then able to make informed decisions based on distributions of kinds of differences. Those decisions might include the choice of candidates for particular refactoring approaches or the evaluation of the effort involved in the refactoring of particular clones.

The discrimination is performed with the algorithm presented in figure 2 where RV, N, M and TE are defined as the sets of differences affecting the signature of methods (return value, name, modifiers, thrown exceptions). Those sets will contain at most one element each. $TypeDiffs$ is the set of differences affecting types and $Other = Diff \setminus (RV \cup N \cup M \cup TE \cup TypeDiffs)$ is the set of all the other differences. Method *choseSet* returns the set of differences ($RV, N, M, TE, TypeDiffs$ or $Other$) corresponding to the type of a difference received in parameter.

Additional information for refactoring
After analyzing the differences, several additional information are computed for each cluster of clones. Those information provide a broader perspective that can be used to assess and compare cloning throughout the system. Let $cloneSize : Clones \longrightarrow \mathcal{N}$ be the size of each cloned method. Also $cloneSize_i = cloneSize(C_i)$. The additional information can then be expressed as:

- The amount of differences between code fragments given by:

$$nbDifferences = card(Diff) \tag{4}$$

- Quantitative aspects of differences: the size of each code fragment involved in a difference, the minimum, maximum and average code fragments involved in a difference. Those sizes are expressed both in lines of code and for a finer measure, in tokens:

$$size : d[i] \longrightarrow \mathcal{N} \; \forall i \in [1..n] \tag{5}$$

$$minsize = min(\forall d : Diff \; \forall i \in [1..n] \; size(d[i])) \tag{6}$$

$$maxsize = max(\forall d : Diff \; \forall i \in [1..n] \; size(d[i])) \tag{7}$$

$$avgsize = \frac{\sum_{\forall d:Diff \;\; \forall i \in [1..n]} size(d[i])}{nbDifferences * n} \tag{8}$$

(9)

- The total amount of lines of code or tokens involved in differences for each cloned method:

$$\forall i \in [1..n] \; totalDiff_i = (\sum_{\forall d:Diff} size(d[i])) \tag{10}$$

- The average proportion of differences compared to the size of the code fragments:

$$\forall i \in [1..n] \; pDiff_i = \frac{totalDiff_i}{cloneSize_i} \tag{11}$$

$$avgPDiff = \frac{(\sum_{\forall i \in [1..n]} pDiff_i)}{n} \tag{12}$$

- The quantity of differences that affect types:

$$nbTypeDiffs = card(TypeDiffs) \tag{13}$$

- The quantity of ordinary differences:

$$nbOtherDiffs = card(Other) \tag{14}$$

- The proportion of differences affecting types:

$$pTypeDiffs = \frac{nbTypeDiffs}{nbTypeDiffs + nbOtherDiffs} \tag{15}$$

Those information will serve in the evaluation of the cloning phenomenon throughout the system and the comparison of clusters of clones from a refactoring perspective. Indeed, they will show the probable effort needed in refactoring. The use of that information will be discussed in more detail in section 5.

The difference analysis, that has been presented in this section, provides detailed information on differences between cloned methods. The meaning of each syntactic difference is expressed in terms of a programming language entity, easily interpretable by a programmer. Such detailed information may facilitate refactoring decisions at the detailed level of the cluster. To gain a broader perspective, difference information is synthesized in two manners. First, differences are grouped according to their impact on possible redesigns. Seven types of differences are distinguished in the analysis but other categories could also be developed. Second, quantitative information on the distributions of differences in clones are computed. Such information allow the analysis of the cloning phenomenon throughout the system and may serve in the selection of candidates for refactoring as well as refactoring approaches to apply.

3 Context analysis

Some refactoring approaches involve the transfer of code fragments between classes thus affecting contextual dependencies. We define the latter as all uses of identifiers (methods or variables) that are neither static nor locally defined in a code fragment and depend therefore on the class containing the method.

In this section, the analysis of contextual dependencies of clones is presented. More precisely, in this second part of the advanced clone analysis, two aspects of context dependent operations are analyzed. First, the list of context dependent operations is computed separately for each clone. This result gives the list and exact number of contextual dependencies in all clones belonging to a cluster.

The contextual dependencies present in the common parts of clones (defined as those not belonging to differences) are then extracted. This measure is interesting as it may influence the choice of refactoring approach by determining the differences in the costs of transferring common code, particular code (the code of the differences) or all the code, between classes. The measure is a conservative estimate of the common context dependencies. Some common source code might indeed be merged with a proximate difference during refactoring and hence diminish the amount of common dependencies.

In this analysis, the list of context dependent operations is computed for each cloned method as follows:

$$\begin{aligned}
&ContextDep = \{\forall i \in [1..n]\ \forall id \in \{Calls \cup Used\} \mid \\
&id \notin Locals(C_i) \wedge (\exists cl : Classes \mid cl = DefiningClass(C_i) \\
&\wedge\, id \in \{Attributes(cl) \cup DefinedMethods(cl)) \bullet \\
&C_i \longrightarrow id\}
\end{aligned} \tag{16}$$

Where:

- $Classes, Methods$ and $Variables$ are respectively defined as the sets of all classes, methods and variables defined in the system.
- $DefiningClass : Methods \longrightarrow Classes$ associates to each method its defining class.
- $Locals : Methods \longrightarrow \mathcal{P}(Variables)$ associates to each method its locally defined variables.
- $DefinedMethods : Classes \longrightarrow \mathcal{P}(Methods)$ is defined as the set of methods defined in a class.
- $Attributes : Classes \longrightarrow \mathcal{P}(Variables)$ is defined as the set of attributes of a class.
- $Calls : Methods \longrightarrow \mathcal{P}(Methods)$ is defined as the set of methods called from within the body of a method.
- $Used : Methods \longrightarrow \mathcal{P}(Variables)$ is defined as the set of variables used within the body of a method.
- $ContextDep : Methods \longrightarrow \mathcal{P}(Calls \cup Used)$ associates to each method a set of identifiers corresponding to context dependent operations.
- $ComContextDep : Clones \longrightarrow \mathcal{P}(Calls \cup Used)$ associates to each cluster of clones a set of identifiers corresponding to context dependent operations common to all clones in the cluster.

The list of context dependent operations common to all clones in a cluster is computed with:

$$\begin{aligned}
&ComContextDep = \{\forall i \in [1..n]\ \forall id \in ContextDep(C_i) \mid \\
&(\nexists d : Diff \mid\ \exists i \in [1..n] \mid id \in d[i]) \bullet Clones \longrightarrow id\}
\end{aligned} \tag{17}$$

This second aspect of the analysis provides a very novel perspective on clones. It determines the coupling strength between shared functionalities and their contexts of use. If the coupling is low (few context dependent operations exist) the transformation of the functionality into a reusable component may be possible without significant overhead. If the coupling is high, but resides in the differences between clones, then encapsulating those differences and decoupling them from the shared code might be the appropriate approach. Finally if high context dependence resides in the common parts of clones, maybe the functionality should be rewritten to decouple the context from the shared behavior. In all situations, the advanced analysis provides most useful input for carrying such redesign decisions.

4 Experiment

The advanced clone analyses presented in the previous sections has been implemented in Java, using JDK 1.1.7. To get the ASTs of the source files, a Java parser generated with Javacc version 0.8 (first pre-release) has been used.

The clones of JDK 1.1.5 [15], a development kit from Sun Microsystems with 145 000 lines of code have been analyzed. The experiment was conducted on a Pentium II 350MHz with 128MB RAM running Linux.

The process used for the experiment is depicted in Figure 3. We have first applied Patenaude et al.'s approach [14] to find clusters of similar methods using metrics. 244 clusters have been found and used as input to the process.

All the information presented in the previous sections has been computed:

- The list of differences between all code fragments.
- The fine-grained difference information: measures of differences considered at the token level. (Equations 4 through 12)
- The detailed difference information comprising the quantitative aspects of differences when considered as lines of code as well as the detailed information on the different types of differences. (Equations 4 through 15)
- Finally, contextual dependencies particular to each clone and common to all clones. (Equations 16 and 17)

Figure 4 shows the distribution of average sizes of cloned methods in the clusters used as input. The amount of clusters decreases with the size of clones (Figure 4) as smaller methods resemble others more easily. Methods having 6 lines of code and less have been removed from clone analysis as for such small methods, clone detection techniques produce an important amount of false positives. Even though some clusters contain particularly long methods, most methods have less than 40 lines.

A first interesting result is presented in Figure 5 where the quantity of clusters of clones is shown for different percentages of method bodies covered by differences.

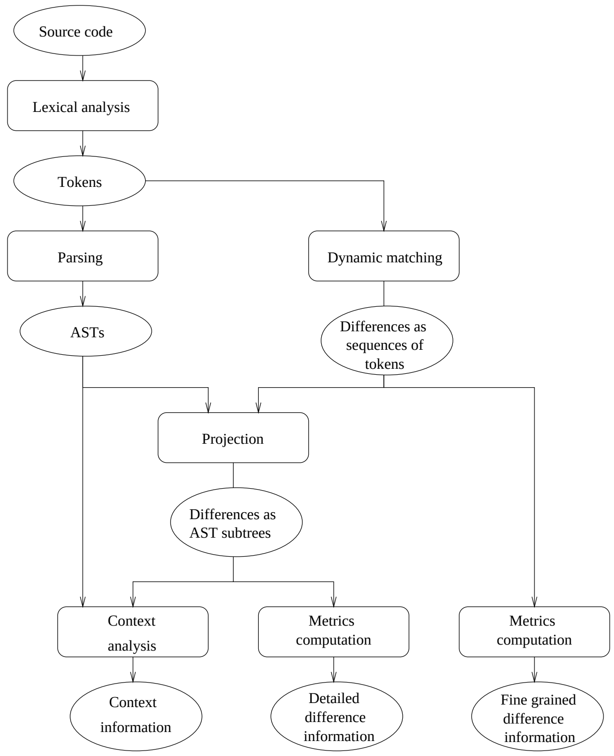

Figure 3: Experimental process.

Table 1: Clusters of clones with differences in the signature of cloned methods

Type of difference	Number of clusters
Return value	20
Method name	56
Modifiers	46
Thrown exceptions	125

The graph shows an increase in the quantity of clusters with the percent of the method covered by differences until 70 percent after which a considerable drop occurs. This drop corresponds to thresholds used during clustering. If we had used smaller thresholds, fewer clusters would have been produced but they would have been more similar. On the other hand, higher cutoff points would have allowed for more clusters with more differences.

Although the clone analysis technique allows the detection of similar code fragments within a given threshold, the amount of clusters containing almost identical clones is high. As many as 22, almost 10% of all clusters, contain methods covered with differences at less than 10% of their size.

The analysis of differences types (Figure 6 and Table 1) shows that many cloned methods (half of the clusters)

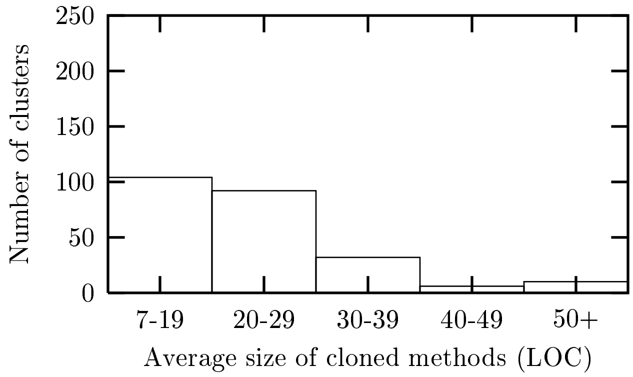

Figure 4: Distribution of average sizes (lines of code) of methods in clusters of clones.

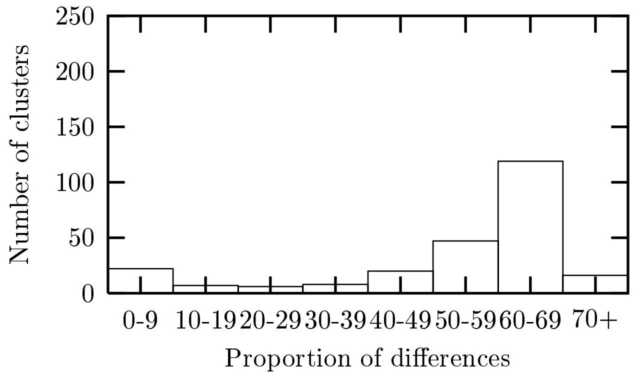

Figure 5: Proportion of clones covered by differences (tokens).

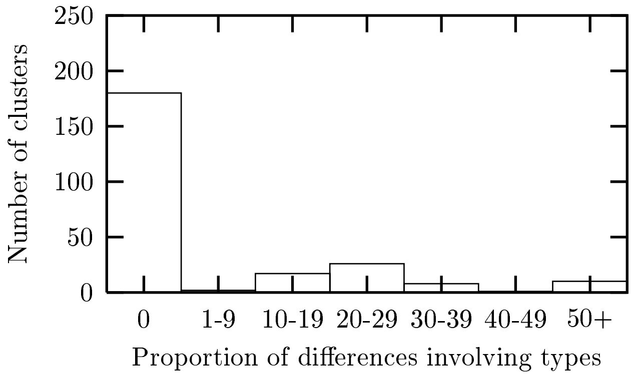

Figure 6: Proportion of differences involving types.

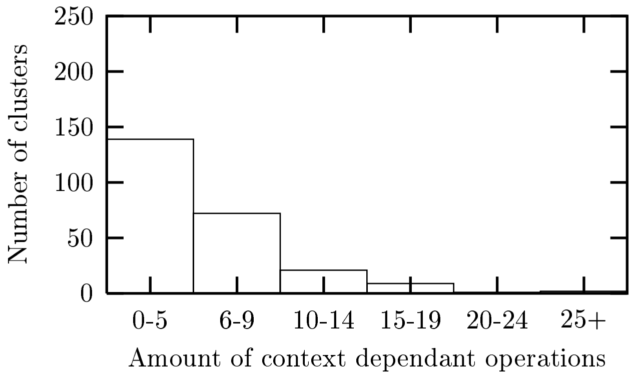

Figure 7: Distribution of the quantity of context dependent operations.

differ in the list of thrown exceptions. Many clones also differ in their name and their modifiers but less in their return values. Most clusters don't contain any differences involving types. Those who do, usually contain several of them, mostly around 20 to 30%

Figure 7 shows the distribution of the quantity of context dependent operations. Most clusters contain few such operations. If a conservative estimate of the operations belonging to common parts of clones is taken, two extremes are obtained (c.f. Figure 8). In most clusters, less than 10 percent of context dependent operations belong to shared source code. In an other important part of the clusters, more than half of context dependent operations are in the common parts of clones.

5 Discussion

The experiment on the analysis of clones in JDK 1.1.5 shows the applicability of our approach to large software systems. 244 clusters of clones corresponding to a little less than 800 methods have been analyzed in less than half an hour.

Clone analysis in JDK

From the results of the clone analysis applied to JDK, several refactoring decisions can be taken. The proportion of differences involving types (Figure 6) shows that in JDK, clone based refactoring could be divided into two phases: one focusing on clusters containing only ordinary differences and the other aimed specifically at clones differing in the types of the data they manipulate. Over 50 clusters of clones contain between 10 and 40 percent of differences explicitly affecting types. This also shows that the division of differences along their types provides a precise refactoring oriented comparison basis for cloned methods. Such comparisons may be helpful in the choice of refactoring candidates and the refactoring actions applicable to them.

From the point of view of the coupling between shared code fragments and their contexts of use, Figure 7 shows that in JDK, most clones contain only few of such dependencies. Clusters may then be divided into two groups for the application of specific refactoring actions. One group with clones containing less than 5 context dependent operations and the other with those containing

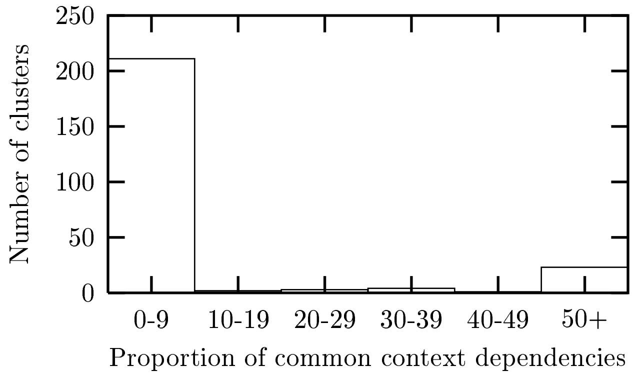

Figure 8: Percent of context dependent operations belonging to the common parts of clones.

more. Refactoring actions involving the transfer of all the code (common and particular) between classes could then be applied to the first group without significant overhead. Refactoring actions keeping as much code as possible in each original class would be preferable for the other group of clones.

Context dependence could guide JDK refactoring decisions even more precisely. Figure 8 shows that the majority of context dependent operations reside in the differences between clones. Therefore, in this system, cloning seems to reuse some functionality and add context dependence to it. The refactoring could then extract the common, loosely couple code fragments, and merge them into a new component while leaving the differences in the original classes. For other systems, other approaches might be more appropriate.

From the results of advanced clone analysis applied to JDK, it can be seen that this analysis is valuable to support refactoring decisions. It can indeed help in the choice of candidates for redesign and their grouping along the refactoring actions that best apply. The detailed lists of contextual dependencies and differences can then be used to guide the actions themselves. The next section discusses the general use of the analysis for refactoring.

Use of clone analysis for refactoring

The first decision while refactoring a system using clone information is the choice of appropriate candidates. Indeed, some clones might belong to sensitive parts of a system and shouldn't be touched, while others might belong to components with a high failure rate, thus more important to redesign. Other characteristics such as effort in refactoring or required refactoring approach may also determine the choice of candidates. The advanced clone analysis presented in this paper is able to guide such decisions by providing detailed information on the degree of similarity of clones. As Figure 5 shows, some clones contain a lot of differences while others are very similar. The degree of similarity is an important information as it corresponds to differences and may hence be proportional to the refactoring effort.

The knowledge of types of differences (c.f. Figure 6 and Table 1) might also influence the choice of candidates. Candidates containing only ordinary differences might be preferred as their transformation should present less difficulties. If a particular refactoring approach is readily available (automated tool, previous experiences, etc.) clones that fit the approach at hand can also be isolated.

Once the candidates have been chosen, an appropriate refactoring process has to be selected. The clone analysis proposed here is also helpful during this phase of redesign. The types of differences as well as the amount of context dependent operations and mostly the percent of those operations belonging to common parts of code will partly determine the most appropriate refactoring. If many context dependent operations belong to differences between clones, it might be best to use an approach that will keep those differences in the original classes, hence allowing to keep the context coupled with the method. If most clones contain differences explicitly affecting types, then an approach allowing the creation of a parent to those types might be most suitable.

The clone analysis proposed can also be used to determine the effort necessary to refactor a cluster of clones. Indeed, an exhaustive list of all differences with their types can be produced. A similar list of contextual dependencies can also be obtained. Not only do those lists present the effort necessary for refactoring, they also correspond to actions that will actually have to be performed.

Computer-aided refactoring process

From the discussion above, it can be seen that the advanced clone analysis provides information useful at different stages of a refactoring process. It could therefore be a good basis for computer-aided object-oriented system refactoring. The analysis could be incorporated in such a process as follows:

- The analysis could first provide general information for the choice of candidates for refactoring.
- After the selection of candidates, it could determine and list applicable redesign approaches, using the characteristics of the selected clusters of clones.
- Once the precise refactoring and the set of candidates would be known, the list of actions to perform could be determined by the tool and provided to the user.
- The programmer could then perform the refactoring using his own judgment to take into consideration any relevant particularity.

Automatic refactoring
We have implemented an automatic refactoring process in CloRT (Clone Reengineering Tool). The process transforms clones along one of two design patterns, *Strategy* [4] and *Template method*. The process factors common parts of clones, parameterizes their differences and decouples their context while producing a configurable and reusable component.

Figures 9 and 10 present the UML diagrams of both approaches. In those diagrams *OriginalClass1* and *OriginalClass2* represent two classes originally containing cloned methods. *CloneHandler* is a new class containing the general code resulting from the factorization of clone commonalities and the factorization of their differences. Classes *ConcreteDiffStrategy1* and *ConcreteDiffStrategy2* of Figure 9 contain the encapsulated differences for each original clone. In a similar manner, classes *ParamCloneHandler1* and *ParamCloneHandler2* of Figure 10 contain the differences of each clone. Interfaces *IdStrategy* and its descendants contain the signatures of all context dependent operations, implemented in the original classes.

The tool, developed with JDK 1.1.7, has been applied to JDK 1.1.5 for the partial redesign of 26 clusters of clones showing that automatic clone based refactoring is possible.

From the experiment we have concluded that an automatic approach could advantageously be enhanced by targeted user interactions. Indeed, the user could better exploit clone particularities for their refactoring and could help alleviate the complex analysis required by an automatic encapsulation of differences. A computer-assisted refactoring approach as described previously allows the user to gain that flexibility while still benefiting from the support of a detailed clone analysis.

6 Related work
Previous research has studied the detection of clones in software systems. Several techniques have been investigated. Some are based on a full text view of the source code. Johnson [9] 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. Ducasse et al. [7] use an exact string matching approach along with visualization for a semi-automatic detection of exact copies of code.

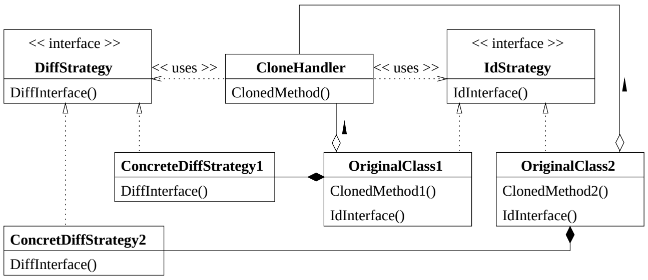

Figure 9: Clone refactoring based on the *Strategy* design pattern.

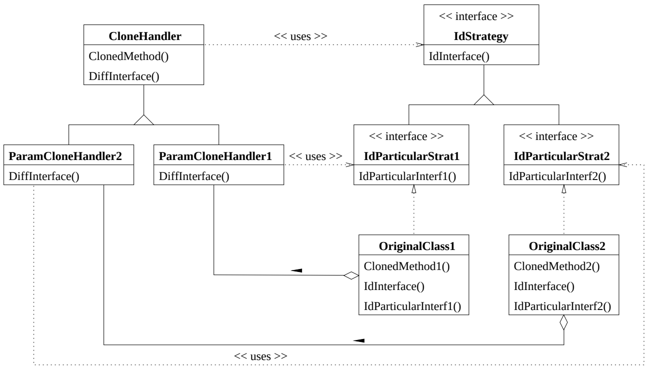

Figure 10: Clone refactoring based on the *Template method* design pattern.

Other approaches, such as those pursued by Mayrand et al. [13] and Kontogiannis et al. [11], 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 [11], 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 of a system. Baxter et al. [5] have investigated this technique.

Several applications of clone detection have also been investigated, Johnson [9] visualizes redundant substrings to ease the task of comprehending large legacy systems. Mayrand et al. [13] as well as Lagüe et al. [12] document the cloning phenomenon for the purpose of evaluating the quality of software systems. Lagüe et al. [12] have also evaluated the benefits in terms of maintenance of the detection of cloned methods.

Merging the common parts of cloned pieces of code has also been investigated. In [5], Baxter et al. use macros to eliminate redundancies and thus reduce the quantity of source code in a system. Although macros are applicable to all detected clones, 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.

Even though those clone analysis studies provide useful information on the cloning phenomenon in software systems, none is oriented towards providing information detailed enough for clone based refactoring.

7 Conclusions and future work

This paper has presented an advanced clone analysis useful to system refactoring. Clones are good candidates for refactoring as they correspond to duplicated code and implicit links between components.

The novel analysis focuses on two aspects of clones: the meaning of their differences from a programmers perspective and their context dependence. Differences interpretation in terms of precise programming language entities is useful for refactoring as it can guide the choice of candidate clusters and guide reengineering actions performed during redesign. For this aspect of the analysis, a novel clone comparison algorithm has been introduced along with a novel difference interpretation and classification. The second aspect of the analysis, context dependence, provides useful input on the cost of transferring common or particular code fragments between their original classes and other classes. It can therefore guide the choice of refactoring approaches to apply to specific clusters of clones.

The novel clone analysis provides a solid basis for the development of computer-aided refactoring environments. Completely automatic approaches are also possible but they are costly and provide little flexibility to the user. Manual refactoring on the other hand, can be cumbersome as hundreds of clusters are present in a system. Assisted refactoring combines the strengths of both approaches by computing all the information necessary for refactoring and allowing the user to concentrate on the refactoring decisions.

The next step of the research is to develop more interactive approaches and investigate their effectiveness in industrial refactoring projects. The refinement in metrics for clone analysis could also be investigated for refactoring performance.

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] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 326–336. IEEE Computer Society Press, October 1999.

[5] 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.

[6] J. Brant and D. Roberts. Refactoring browser. st-www.cs.uiuc.edu/~brant/RefactoringBrowser. http://st-www.cs.uiuc.edu/ brant/RefactoringBrowser/.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance 1999*, pages 109–118, 1999.

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

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

[10] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. *Proceedings of the 4th Working Conference on Reverse Engineering*, pages 44–54, 1997.

[11] 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.

[12] 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.

[13] 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.

[14] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th. International Workshop on Program Comprehension. IWPC'99*. IEEE Computer Society Press, 1999.

[15] Sun Microsystems Inc. Jdk 1.1.5.: Java development kit.