Detecting Higher-level Similarity Patterns in Programs

Hamid Abdul Basit
Department of Computer Science
School of Computing
National University of Singapore
+65 6874 2834
hamid@nus.edu.sg

Stan Jarzabek
Department of Computer Science
School of Computing
National University of Singapore
+65 6874 2863
stan@comp.nus.edu.sg

ABSTRACT
Cloning in software systems is known to create problems during software maintenance. Several techniques have been proposed to detect the same or similar code fragments in software, so-called *simple clones*. While the knowledge of simple clones is useful, detecting design-level similarities in software could ease maintenance even further, and also help us identify reuse opportunities. We observed that recurring patterns of simple clones – so-called *structural clones* - often indicate the presence of interesting design-level similarities. An example would be patterns of collaborating classes or components. Finding structural clones that signify potentially useful design information requires efficient techniques to analyze the bulk of simple clone data and making non-trivial inferences based on the abstracted information. In this paper, we describe a practical solution to the problem of detecting some basic, but useful, types of design-level similarities such as groups of highly similar classes or files. First, we detect simple clones by applying conventional token-based techniques. Then we find the patterns of co-occurring clones in different files using the Frequent Itemset Mining (FIM) technique. Finally, we perform file clustering to detect those clusters of highly similar files that are likely to contribute to a design-level similarity pattern. The novelty of our approach is application of data mining techniques to detect design level similarities. Experiments confirmed that our method finds many useful structural clones and scales up to big programs. The paper describes our method for structural clone detection, a prototype tool called Clone Miner that implements the method and experimental results.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement – *Restructuring, reverse engineering, and reengineering*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *Clustering.*

ESEC-FSE'05, September 5–9, 2005, Lisbon, Portugal.

General Terms
Algorithms, Design.

Keywords
Software clones, similarity patterns, clone detection

1. INTRODUCTION
Software maintenance is widely accepted as the most costly phase of a software lifecycle, with figures as high as 80% of the total development cost being reported for it [41]. Cloning is one of the contributors towards this cost.

In the past decade, clone detection and resolution has got considerable attention from the software engineering research community and many clone detection tools and techniques have been proposed [5][7][11][12][13][21][22][28][30][33]. So far, clone detection has been focused on detecting similar code fragments – so-called simple clones, with some gains in reducing update anomalies and the software size. These gains, however, can be improved by elevating the level of clone analysis. As shown by our previous studies [10][4], clone analysis aided by domain analysis can reveal design level similarities (so-called structural clones), whose unification not only brings more size reduction, but also helps in understanding the design of the system for better maintenance and future enhancement.

The work presented in this paper is the first of its kind in analyzing patterns of cloned fragments to infer design-level similarities in a system. We started by formulating heuristics to characterize patterns of simple clones that could indicate design-level similarities. The data resulting from the detection of simple clones in big software systems can be large and complex, making manual application of heuristics hardly possible. We applied the data mining approach to extract the most useful information leading to the detection of closely interacting classes and modules, as candidates for structural clones. Our experimentation has so far focused on detecting some basic, but useful, types of design-level similarities such as groups of highly similar classes or files. The principle of our method, however, scales to finding many other types of similarity patterns.

The original contribution of our work lies in formulating heuristics for inferring design-level similarities based on patterns of simple clones, and applying data mining approach to automate making proper inferences. We also demonstrate that our method finds useful design-level similarities and scales up to large programs. Finally, our method also allows us to find complex gapped clones (e.g., code fragments that differ in arbitrary

number, location and size of added/deleted code fragments), which extends capabilities of existing token-based clone detection techniques and tools that can find exact and parameterized clones only.

We apply token based approach for the detection of simple clones. It provides a suitable level of flexibility for the task by limiting the language dependence, being resilient to the differences in code layout, while providing a good mechanism for detecting parameterized simple clones. Having transformed a source program into a string of tokens, we compute the maximal repeats in the string with a suffix array based algorithm [1]. These maximal repeats, with some heuristic based pruning, form clone classes. Although our detection of simple clones is much similar to the previously published approach [22], the novel contribution is in the introduction of a simple and flexible tokenization technique, and the selection of efficient data structures and algorithms for token string manipulation.

While clone analysis based on different metrics calculated for the clones has been applied previously [22], the idea of clone pattern mining is our original contribution. We can find clone patterns in different units of code, either methods or classes or components or modules, gaining useful insights into the cloning situation at different levels of abstraction. We have initially tried this approach at file level, by finding the frequently occurring clone patterns in different files and analyzing those patterns, with promising results.

By detecting the frequently co-occurring clone classes in different files, we can isolate the groups of files that have strong similarity with each other. This is achieved by a clustering algorithm that we have devised for this particular problem. These clusters of highly similar files form basic structural clones.

The remainder of this paper is organized as follows. After describing the cloning problem in Section 2, we give the motivation for Clone Miner in Section 3. Sections 4 and 5 discuss our method for detecting simple and structural clones, respectively. Section 6 describes the experimentation with the prototype tool, while Section 7 gives the implementation details. Related work and conclusions end the paper.

2. THE CLONING PROBLEM

Two code structures of considerable size are clones of each other if there is significant similarity between them. The actual size and similarity (which can be measured, for example, in terms of percentage of repeated code) varies depending on the context, and is left to human judgment. Clones may or may not represent program structures that perform well-defined functions. The above notions involve human judgment and are, therefore, subjective in nature. Unfortunately, similarity is a multi-faceted phenomenon that escapes precise definition.

Cloning is a common phenomenon found in almost all kinds of software systems. Recently, this phenomenon has caught considerable attention from the research community. Cloning is believed to have a negative impact on the maintenance of large software systems.

Most of the interesting clones, particularly those at the higher level (so called structural clones), are similar but not identical. Changes among clones result from differences in intended behavior, and from dependencies on the specific program context in which clones are embedded (such as different names of referenced variables, methods called, or platform dependencies).

Reuse in object-oriented systems is made possible through different mechanisms such as inheritance, shared libraries, object composition, and so on. Still, programmers often need to reuse components which have not been designed for reuse. This may happen during the initial systems development and also when these software systems go through the expansion phase and new requirements have to be satisfied. In these situations, the programmers usually follow the low cost copy-paste technique, instead of costly redesigning-the-system approach, hence causing clones. This type of code cloning is the most basic and widely used approach towards software reuse. Several studies suggest that as much as 20-30% of large software systems consist of cloned code [5][33].

Programmers may also clone code to speed up development and maintenance, especially when the new requirement is not fully understood and a similar piece of code is present in the system. Cloning may also be linked to LOC-based performance appraisals and the fact that cloning is considered safe as having little unplanned effect on the original code, as the original code is not modified and simply copied at another place. Sometimes, cloning is done to increase the robustness of life-critical systems, for better performance, or to minimize dependencies among developers in large projects, or to port the application to another hardware platform. Poor design and ad hoc maintenance also induce clones. Clones are also created by code generation tools, or by following a coding style. Finally, some clones may just appear accidentally.

While there are good reasons for creating certain clones, most of them, independently of the reasons why they occur, are counter-productive for future maintenance, as they increase the risk of update anomalies (inconsistencies in updating clone instances). When a cloned fragment is to be changed, a programmer must find and update all the instances of it consistently. The situation is further complicated if an affected fragment must be changed in slightly different ways, depending on the context. With excessive cloning, evolution and further development (Maintenance) become prohibitively expensive. Clones may also form implicit links between components that share some functionality. All this contributes towards "software aging" [36].

We distinguish two types of clones, namely:

Simple clones: contiguous segments of similar code such as class methods, or fragments of method implementation, and

Structural clones: patterns of inter-related classes emerging from design and analysis spaces; patterns of components at the architecture level; design solutions repeatedly applied by programmers to solve similar problems (so-called "mental templates" [7]).

Different types of simple clones have been discussed in literature. An *exact clone* is a fragment of code that is identical to another one. *Parameterized clones* are defined as "code sections that match except for a one-to-one correspondence between candidates for parameters such as variables, constants, macro names, and structure member names" [6]. Some other authors do not consider strict one-to-one relationship between the parameters of two cloned portions and their treatment of parameterized clones is more general [22]. A *gapped clone* is a code fragment that is an

exact or a parameterized clone of another, but has some extra or missing code that cannot be parameterized.

The clones discussed above are simple code level clones that consist of a single chunk of code cloned at different places, with the exception of gapped clones, which can also be a manifestation of the structural clones, as will be explained later.

Recurring problems of a similar structure in analysis and design spaces may lead to structural clones. Analysis patterns [14] and design patterns [15] exemplify these situations. Structural clones often represent repeated patterns of simple clones. For example, Figure 1 shows a pair of structural clones our industry partner (SES Systems Pte Ltd) found in a real C# system. These clones arose from the following situation: The system was based on over 20 domain entities such as User or Task. The design and implementation of operations (such as Create) for various entities were characterized by a similar pattern of collaborating classes across GUI, service and database layers. Each box in Figure 1 represents a number of classes, with much similarity across classes implementing similar concepts in the same types of operations for different entities.

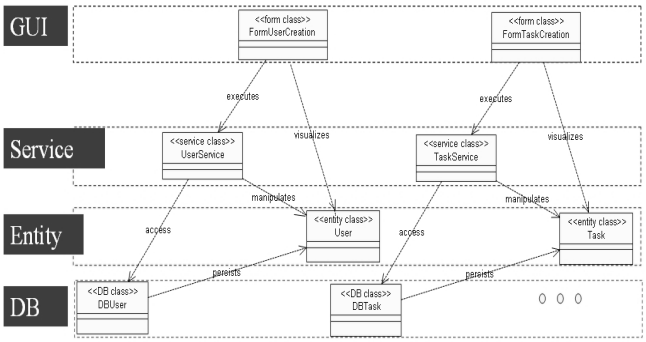

Figure 1. A pair of structural clones

Despite striking similarities, SES could not design a generic solution for groups of operations such as CreateUser and CreateTask. To implement generic operations for domain entities, SES would have to first unify groups of classes in GUI, Service, Entity and DB layers such as Create[entity-type]Form, Update[entity-type]Form, etc. However, the nature of variations in business logic across operations for different entity types was such that neither inheritance nor type parameters (such as in generics proposed for C# [25]) could be used to implement operations in generic way. Therefore, in the C# subsystem, SES had to repeat the same design/implementation steps for all the domain entities and their respective operations.

3. MOTIVATION FOR CLONE MINER

Our requirements for clone detection differ from other tools and techniques. We wish to detect both simple and structural clones (such as the one shown in Figure 1), allowing a vast range of differences between them.

Simple clones that are of our interest can differ in type parameters, keywords, variable/constant names, operators – actually any details of algorithms, declarations or function signatures. In particular, two similar fragments might be edited by a programmer in arbitrary ways, e.g., by modifying some section, or inserting/deleting certain lines of code.

The simplest form of a structural clone is a class. We are interested in classes differing in details of method implementation, method signatures, or the order in which methods are listed in the class body. If a class contains extra methods as compared to other similar classes, we could still consider such classes as structural clones. Beyond similar classes, we are also interested in patterns of classes/components that display similarities.

The reason why we are interested in such vast range of similarities is that our goal for clone detection is to unify clone classes with generic design solutions. We build generic design solutions with a meta-level method, called XVCL [42], which is capable of unifying both simple and structural clones, even with a wide range of differences among them [4][18]. With XVCL meta-structures, we would like to unify any similarity patterns whose unification is deemed beneficial from the engineering point of view (e.g., leads to simpler programs that are easier to maintain or reuse due to non-redundancy).

Existing techniques for clone detection solely focus on simple clones. Furthermore, these techniques can only find exact clones or clones differing in parametric ways.

Our goal is to overcome these limitations. We expect Clone Miner to detect candidates for simple and structural clones that meet certain similarity threshold (as explained later in the paper). Such clone candidates would be examined by an analyst who would then decide which ones are suitable for unification with generic design solutions. Ideally, information about accepted clones could be fed to the Clone Miner for further processing, in order to find higher-level clones based on lower-level ones. Therefore, the clone detection process that we envision would lead from simple clones to structural clones, possibly at a number of abstraction levels (up to the level of software architecture).

4. DETECTION OF SIMPLE CLONES

In Clone Miner, we have implemented the token based detection of simple clones. Although the token similarity computations are expensive, yet token based approach gives certain advantages over other techniques.

With negligible amount of pre-processing, clone detection based on raw text is language independent and free of pre-processing overhead, but is very sensitive to the small differences that may be present between two very similar code fragments. It can only find the exact clones, and cannot be used for parameterized clone detection.

On the other hand, parse tree or syntax tree comparison based clone detection becomes too language dependent, and strongly tied to the specific flavor of the given language. Similarly, metrics based techniques are also language dependent but their major restriction is on the granularity of the clones i.e., only functional or class level clones can be detected. Hence we have chosen the token-based approach, that lies somewhere in the middle of the pre-processing spectrum.

After tokenizing the input source code files, a single token string is generated. Efficient suffix array based repeat finding algorithm, with some language-specific, heuristics based optimizations, is implemented to detect clones.

4.1 Tokenization

As explained previously, our requirements for clone detection differ from the other tools and techniques. Hence, we propose a customizable tokenization strategy. In this scheme, a separate integer ID is assigned to each token class found in the source code. The classification of tokens is totally customizable. For example, if the user does not want to differentiate between the types {***int, short, long***, ***float, double***}**,** we can have the same ID to represent every member of the above set of types.

An important parameter that the user needs to adjust for clone detection is the minimum length of the clones. If this threshold value is too large, few clones are reported. On the other hand, if the threshold value is too small, a large number of clones are reported, many of them being so small that no meaningful clone unification can be applied. In CCFinder [22], the default value is set to 30 tokens, but in Clone Miner, it is advised to set a smaller threshold value to facilitate the detection of structural clones. Several small clones may form a bigger gapped clone, having multiple gaps of arbitrary sizes, and it will be reported as a clone pattern in the next stage of clone pattern mining.

The sequence, in which the input files are read in, is not relevant to clone detection. Clones crossing these file boundaries are not meaningful in terms of clone unification as this ordering may be random and with another ordering of the input files, such clones may not be detected. To curb this straddling of files boundaries by clones, the boundaries for files are marked by unique sentinel tokens that are never repeated in the token string. This ensures that the clones never cross a file boundary. These sentinel tokens increase the alphabet size, but the string algorithms are so selected that this effect is only marginal. Detection of file boundaries is straightforward as each file is read in separately.

Method boundaries are also detected by an online finite state machine (FSM), but they are not marked, since we are not restricted by method boundaries for clone unification using XVCL. Several small methods may form a bigger clone, which may not be detected if the method boundaries are marked by unique sentinel tokens. We plan to use this functionality in future for method based clone pattern analysis.

4.2 Finding Clones

Here we reproduce the definitions of a few important clone related terms given in [22]:

"A ***clone relation*** is an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code portions. A clone relation holds between two code portions if (and only if) they are the same sequences. For a given clone relation, a pair of code portions is called ***clone pair*** if the clone relation holds between the portions. An equivalence class of clone relation is called ***clone class***. That is, a clone class is a maximal set of code portions in which a clone relation holds between any pair of code portions." (We have used the terms *clone* and *clone classes* interchangeably whenever it did not lead to confusion.)

The detection of all clone classes corresponds directly with the computation of all non-extendible 'repeats' in a string, when the code is represented by a string of tokens. The sentinels for method and file boundaries make sure that no 'repeat' crosses these boundaries. The algorithm for finding these non-extendible 'repeats' returns the output in terms of the indexes in the token string for beginning and ending of the repeat. This information has to be translated into file name, line number and column number to be useful for the user and to be projected on the source code browser. For this purpose, information for line number and column number for each token is stored separately.

The basic output from the Clone Miner gives the number of total clone classes found and the details for each clone class. This includes its length in tokens, number of clone instances (members of this clone class), file ID for each clone instance, its beginning line number and column number, and its ending line number and column number. A sample is shown in Figure 2.

CLONE CLASS ID : 24

LENGTH : 32 TOKENS

MEMBERS : 2

FILE ID	START LINE	START COLUMN	END LINE	END COLUMN
10	462	19	464	5
15	894	18	896	5

Figure 2. Basic clone class information

This means clone class 24 is 32 tokens long and has 2 members. The first member is present in file 10 starting at line 462 and column 19 and ending at line 464 at column 5. The second row is interpreted in a similar way. The information about the column numbers is very useful for the Clone Miner GUI as it gives the exact location of the beginning and ending of the clone instance in the file.

Initialization lists for tables and arrays in Java have to be ignored completely as they serve no purpose in clone unification. They are simply lists of different values that are mistakenly detected as clones because they represent the same token class (e.g., integer values). For detecting these initialization lists, we just need to detect beginning token sequence "={" and its corresponding ending braces "}", which can be easily done by a small finite state machine working online, while the input source files are being read.

5. MINING FOR STRUCTURAL CLONES

Having detected individual clone classes, we move on to the detection of higher level similarity patterns based on the data gathered in the previous stage. The first step is to represent this data in a suitable format to facilitate comparison of different bigger units of code like files or classes. Since, in Java, each file contains one class (considering the nested and inner classes as part of the outer class), currently, we only perform this comparison based on files.

By finding clones that occur together frequently in different files, we get more insight into the cloning scenario of the system under analysis. Then we extract significant clone patterns for further detailed manual analysis. These steps are discussed next.

5.1 Finding Frequent Clone Patterns

Once the simple clones have been detected, the data is transformed into a format that lists down all the clone classes represented in each file. This information is presented to the user. But more importantly, this information is used as the input for the

data mining phase of detecting structural clones. A sample extract of this format is shown in Figure 3. The first row says that the file with file ID 12 contains three clone instances belonging to clone class 9 and one instance from each of the clone classes 15, 28, 38, and 40. The interpretation is likewise for the other rows.

FILE ID	CLONE CLASSES PRESENT
…	…
12	9, 9, 9, 15, 28, 38, 40
13	12, 15, 40, 41, 43, 44, 44
14	9, 9, 9, 12, 15
…	…

Figure 3. Clones per file

The first stage of detecting structural clones is the detection of recurring patterns of simple clones in different files. Here, we apply the "market basket analysis" technique from the Data Mining domain. The idea behind this technique is to find the items that are usually purchased together by different customers from a departmental store. The input database consists of a list of transactions, each one containing items bought by a customer in that transaction. The output consists of identifying those groups of items that are most likely to be bought together. The analogy here is that the files represent the transactions and the clone classes represented in that file are the items of that transaction. Our objective is to find all those groups of clone classes whose instances occur together in different files. These patterns of clone classes will act as the unique representation for a group of files, and depending upon its significance in terms of files' coverage, will lead to identifying groups of highly similar files. This will be our basic structural clone.

```
ALGORITHM FOR FINDING FILES CONTAINING
          THE CLONE PATTERN

For each clone pattern:
1. Take the first clone class
2. List all files that contain
   instances of this clone class
3. For each clone class:
       a. If it does not contain all
          the clones of the pattern,
          prune it from the list
4. Output the final list
```

Figure 4. Algorithm for finding files having the clone pattern

Market basket analysis is based on "frequent itemset mining (FIM)". The difference between our data and the expected data format for frequent itemset mining is that in FIM, the items in a transaction are considered unique, whereas in our data, one file may contain multiple instances of a clone class. There can be the possibility to normalize the data, by removing the duplicates, but by doing so, we miss out important information, as multiple occurrences of the instances of the same clone class in different files is a valid pattern of clones. For example, 9, 9, 9, 15 is a valid clone pattern represented in File 12 and File14 in Figure 3.

To normalize the data, Clone Miner transforms the data to make all the clones present in a file unique. For this, our technique is based on the following heuristic assumption; in a given file, not more than N instances of a clone class can be present. We multiply each clone class by N and add the index number of the clone in this file to the result. For example, in the case shown in Figure 3, the transformed data will be like Figure 5 with N = 100.

FILE ID	CLONE CLASSES PRESENT
…	…
12	900, 901, 902, 1500, 2800, 3800, 4000
13	1200, 1500, 4000, 4100, 4300, 4400, 4400
14	900, 901, 902, 1200, 1500
…	…

Figure 5. Transformed clones per file

Now the result from FIM will include 900, 901, 902, 1500, which, after reverse transformation of dividing by 100 and rounding down the result, becomes 9,9,9,15, i.e., reflecting the exact scenario.

Another observation here is that mining all frequent itemsets returns many frequent itemsets that are subsets of bigger frequent itemsets. So the correct solution in our case is to perform "Frequent Closed Itemset Mining" (FCIM), where only those itemsets are reported which are not subsets of any bigger frequent itemset.

Because of the nature of the data mining problem, the algorithm is tuned to adjust the minimum support level for FCIM. For our problem, we have hard coded this to be 2, so that it will report a clone pattern, even if it is present only in 2 files as it could be significant for maintenance based on its size.

FREQUENT CLONE PATTERN	SUPPORT
9,9,9,15	2
60 44 40 42 3 49 59 63	4
…	…

Figure 6. Sample frequent clone patterns with SUPPORT

The output from this stage is in the format shown in Figure 6. Each row represents one clone pattern along with its support count, indicating the number of files containing this clone pattern. In Figure 6, the first clone pattern is present in 2 files, whereas the second one is present in 4 files.

Although this information is very useful, one important piece of information missing here is the identification of those files that contain these patterns. With some extra processing, Clone Miner finds this information as well. The algorithm is presented in Figure 4.

5.2 Clustering Highly Cloned Files

The recurring patterns of simple clones found in the previous stage may or may not be significant. The significance of the clone patterns is subject to the user's judgment. A lot of clone patterns found by the above method may not be significant in terms of file coverage, and there could also be multiple clone patterns represented in a single file. However, all of these clone patterns are still useful, as each one represents an unrestricted gapped clone, where any number of gaps are allowed with arbitrary size and ordering.

Some of these clone patterns could be quite significant and cover a considerable part of some files. Such clusters of files that are

covered by a significant clone pattern form the basic structural clones, and may also indicate higher design level similarities that can be extracted with proper domain analysis.

To measure file coverage by a clone pattern, we calculate two metrics, namely the File Percentage Coverage (FPC), which indicates the percentage of a file covered by a clone pattern, and the File Token Coverage (FTC), which tells the number of tokens in a file covered by the clone patter, for each file containing the clone pattern. The complication here is that some clones may overlap in a file, so we cannot simply add up the size of all clones in a pattern to find the file coverage. The algorithm currently implemented in Clone Miner makes use of a bit vector equal to the size of the file (in tokens), with all bits initially set to false. For each clone of the clone pattern, the corresponding bits in the token file are set and the final count of the set bits gives the file coverage. This is an expensive processing in terms of time because each clone pattern may have several clones and each clone pattern may be represented in several files. It also involves a lot of redundancy as the same clone may be a part of several clone patterns and the same file may be processed again and again. We plan to do some optimizations to this algorithm in future. The output is in the format given in Figure 7.

The first two rows in Figure 7 depict the clone pattern as explained above. The last two lines give the file ID with the FPC and FTC values for each file.

CLONE PATTERN		9,9,9,15
SUPPORT		2
FILE ID	**FTC**	**FPC**
12	1175	29%
14	1175	50%

Figure 7. Frequent clone pattern with file coverage

We now have to decide which clone patterns among those found previously signify clusters of highly similar files. The first consideration is again to represent the data in a suitable format. Instead of representing files, we start by representing clusters. We do not expect that all files may be part of some high level structural clone and should be detected as part of a cluster. Hence a lot of files may have to be ignored as outliers when clustering. This is different from other approaches, where it is assumed that a majority of data points belong to clusters and a few would be detected as outliers. This approach is also called cluster mining as compared to clustering.

We start by considering each clone pattern as a different cluster, where the clone pattern acts as the description of the cluster. The FPC and FTC indicate the significance of each cluster. The user specifies a minimum FPC and FTC value to indicate the significance of a cluster. The cluster will be considered significant even if one file has the FPC or FTC value greater than threshold values. The expected output is to find all the significant clusters that cover maximum number of files and no file is preferably repeated in two clusters.

Our clustering algorithm only involves iterative pruning of clusters as shown in Figure 8.

6. EXPERIMENTATION

We performed a number of tests on different parts of J2SE 1.5 [19] source code with interesting and useful results. One detailed analysis of the Java Buffer Library (java.nio.* package of J2SE 1.5) with Clone Miner is presented here. In previous case study [10], we have already analyzed the structure of this library in detail and have found considerable code and design level cloning in it. Therefore, by choosing java.nio.* package, we could easily validate if the results of automatic clone detection by Clone Miner were relevant, that is, if they revealed significant and useful similarities.

```
      ALGORITHM FOR CLUSTER PRUNING

REQUIRE : initial clusters
INPUT   : min FPC, min FTC

1. Prune all the clusters with FPC of
   all files < min FPC and FTC of all
   files < min FTC
2. Sort clusters based on the SUPPORT
      a. Secondary sort on max. FPC of
         all constituent files when
         SUPPORT is same
3. Starting from the smallest cluster,
   Prune clusters whose constituent
   files are all present in another
   cluster
4. Starting from the smallest cluster,
   prune clusters whose constituent
   files are all present in any other
   cluster.
```

Figure 8. Algorithm for cluster pruning

A buffer contains data in a linear sequence for reading and writing. Buffer classes differ in features such as a buffer element type, memory allocation scheme, byte ordering, and access mode, as shown in Table 1. Each legal combination of features yields a unique buffer class. That is why, even though all the buffer classes play essentially the same role, there are 74 classes in the Java Buffer library.

Table 1. Features in the Buffer Library

LEVEL IN CLASS HIERARCHY	**FEATURE DIMENSION**	**FEATURES**
level 1	buffer data element type	byte, char, int, float double, long, short
level 2	memory allocation scheme	direct, nondirect
level 2	byte ordering	native, non-native, Big_endian, Little_endian
level 3	access mode	writable, read-only

With manual domain analysis, it was found that the 71 buffer classes (all classes except Buffer, MappedByteBuffer, StringCharBuffer) can be clustered into 7 groups of similar classes. Members from each of these groups combine to form a

repeating structure of collaborating classes (a structural clone). The description of these groups is as follows [10]:

1. [T]Buffer: contains 7 buffer classes of type T. T denotes one of the buffer element types, namely, Byte, Char, Int, Double, Float, Long, Short.
2. Heap[T]Buffer: contains 7 Heap classes of type T.
3. Direct[T]Buffer[S|U]: contains 13 Direct classes. U denotes native and S - non-native byte ordering.
4. Heap[T]BufferR: contains 7 Heap read-only classes.
5. Direct[T]BufferR[S|U]: contains 13 Direct read-only classes.
6. ByteBufferAs[T]Buffer[B|L]: contains 12 classes providing views T of a Byte buffer with different byte orderings. T here denotes buffer element type except Byte. B denotes Big_endian and L - Little_endian byte ordering.
7. ByteBufferAs[T]BufferR[B|L]: contains 12 read-only classes providing views T of a Byte buffer with different byte orderings (B or L). T here denotes buffer element type except Byte. B denotes Big_endian and L - Little_endian byte ordering.

By performing the simple clones' analysis of the Buffer Library with Clone Miner, a total of 102 clone classes were detected, with the minimum clone size of 20 tokens. As expected, the clones were distributed in all the files.

46 clone patterns were discovered by the FCIM algorithm when it was run on the data file containing the clones listed in terms of files. These patterns formed the initial 46 clusters. In the first iteration of the cluster pruning phase, 18 insignificant clusters were pruned leaving 28 significant clusters, each having at least one file that is covered more than 50% by the clone pattern describing that cluster. In the second iteration, 13 of the significant clusters were also pruned because all their constituent files were totally represented in some other cluster. After the third and final iteration, another 8 clusters were pruned because the files representing them were present in other clusters as well, leaving behind only 7 clusters. These 7 clusters match exactly with the 7 groups of similar classes that were found manually as mentioned before. This means that by performing only minimal domain analysis based on the output from Clone Miner, the generic representation of the whole Buffer Library can be formulated.

We also analyzed other parts of the J2SE 1.5 with Clone Miner with some very useful results. Table 2 presents a summary of the significant basic structural clones that were found in J2SE 1.5 with different values of FPC and FTC. A manual inspection of the results indicates that the file clusters found by Clone Miner actually indicate significant cloning between the files clustered together. Most of the times, the design similarity was evident from the file names. For example, the files

- OpenMBeanAttributeInfoSupport.java (FTC 1305 tokens, FPC 87.6427%) and
- OpenMBeanParameterInfoSupport.java (FTC 1275 tokens, FPC 91.5948%)

are detected as a cluster in package javax.management. openmbean.*. The cloning in the files could be easily verified with manual inspection. Although other tools with metrics value comparison for different files may also detect such groups of files, the advantage of Clone Miner is that it builds this information incrementally, and at each step all the previous-level information is also available that gives the details of cloning. While comparing the above two files, for example, the user also knows which chunks of code is cloned between the two files and what is the location of those clone instances in each file. A GUI for Clone Miner, that is an ongoing project, will greatly enhance the usefulness of this feature.

Similarly, the 14 files with the same name of ObjectFactory.java, that are duplicated intentionally for each JAXP sub package in com.sun.org.apache.* package are also detected by Clone Miner as a cluster of 14 files having an FPC of almost 98% and an FTC of 1319 tokens each. The fact that the files are in fact exact duplicates of each other is revealed upon reading the comments inside the files.

Further analysis may lead to the detection of even higher-level similarities between the different clusters detected by the tool and is part of our future work.

Table 2. Clone cluster analysis of J2SE 1.5

MIN. FPC	MIN. FTC	NO. OF CLUSTERS	NO. OF FILES
50%	35	471	2620
50%	500	95	298
50%	1000	45	185
90%	35	188	1012
90%	500	42	183
90%	1000	23	95

The results shown in Table 2 indicate that significant gains can be achieved by unifying groups of highly similar classes at the meta - level, both in terms of code reduction as well as in terms of design understanding. Properly representing groups of similar classes, with clearly identifying the similarities and differences, helps greatly in better understanding the design and facilitate further changes, easing the maintenance.

7. TOOL IMPLEMENTATION

The Clone Miner is implemented in C++ with the extensive usage of container classes from STL for efficient manipulation of data.

The lexical analyzer for Clone Miner is built with ANTLR [3]. Only one pass over the source code is required to tokenize the input source code.

We achieve the variable tokenization flexibility in the reverse way. Our lexer generates a very refined tokenization with 115 token classes for Java source code, and then allows the user to specify the equivalence of the different input tokens to facilitate detection of parameterized clones. Since the clone detection algorithm only detects exact string repeats, the parameterized clones are converted to exact clones by tokenization and token equalization. Primitive Java types like int, float, long, double can be easily parameterized and hence should be treated as same. Similarly other possibilities are operators like +,-, /,* etc. and keywords like 'public' or 'private'.

The user can also choose to suppress some token classes. For example, if keywords like 'const' are to be ignored, their respective token number will not appear in the token string.

The detection of the short repetitive fragments, as discussed later in the related work section, is done before running the repeat finding algorithm. It involves reading the suffix array and computing the difference between each pair of consecutive entries of the suffix array. If this distance is being duplicated for some pairs, we mark it as a repetitive section. This section continues until the distance between the consecutive pairs of suffixes of the string remains the same. A bit-vector is used to store this information. For every token that is a part of a repetitive section, the bits are set, while for the rest of the code, the bits are unset. When the clones are being detected, the sections of tokens corresponding to set bits in the bit-vector are ignored.

We have chosen the best algorithms available for the different tasks in the system. The system is designed in such a way that it works with the standard algorithms. Different implementations of these standard algorithms can be simply plugged into the system with minimum configuration. We have incorporated the string matching based on suffix arrays instead of suffix trees considering their space efficiency [32]. Although there are 3 new algorithms published in 2003 for linear time construction of suffix array [23][26][26], empirical studies have shown that they are not as fast in practice as the previously known O(n log n) algorithm [31] which we have chosen for our system. These results are presented in [38][39]. For efficiently finding the repeating parts of the string, the suffix array has to be enhanced by a supporting array called the Longest Common Prefix (LCP) array. For LCP array creation, we are using the algorithm "GetHeight" presented in [24].The algorithm "NERF" (Non-Extendible Repeats Finder) used for non-extendible repeats finding is similar to the maximal pairs finding algorithm presented in [1]. For FCIM, currently we are using the algorithm from [16].

We performed several trial runs of Clone Miner on full J2SE 1.5 source code consisting of 6558 source files and around half a million LOC, on a Pentium IV machine with 3.0 GHz processor and 1 GB RAM. Each time it took less than 3 minutes to run the whole process from finding simple clones until the final cluster pruning. A total of 22,221 clone classes were detected with the minimum clone size of 35 tokens. These were condensed into 13,262 clone patterns, from which we can select the significant ones depending upon the threshold values for FPC and FTC after the 3 layer pruning.

8. RELATED WORK

Different techniques have been deployed for the detection of simple clones. They can be broadly categorized based on the program representation and the matching technique. For program representation, the different options are plain text [13][21], tokens[5][22], abstract syntax trees[7], program dependence graphs [29][30], and metrics for code structures [28]. The different matching techniques include suffix tree based token matching [22], text fingerprints matching [21], metrics value comparisons [28], abstract syntax trees comparisons [7], dynamic pattern matching [28] and neural networks [12].

Our clone detection technique is similar to the one employed by CCFinder [22]. Many of the heuristics used in CCFinder are implemented in Clone Miner as well.

CCFinder is easily configurable to read input in different programming languages like C, C+, Java and COBOL. A suffix-tree matching algorithm is used for the discovery of clones. Some optimizations are applied to reduce the complexity of token matching algorithm.

These optimizations include the alignment of token sequences to begin at tokens that mark the beginning of a statement like #, {, class, if, else etc. Considering only these tokens as leading tokens reduces the resulting suffix tree size to one-third. Another optimization is the removal of short repeated code segments from the input source like case statements in switch-case constructs and constant declarations. Very large files are truncated and "divide-and-conquer" strategy is applied to find duplicates in them.

In contrast to Clone Miner, CCFinder finds the clone pairs and then merges them to form clone classes. In Clone Miner, instead of maximal pairs, the maximal repeats are found directly, with similar space and time complexity. This saves the overhead of forming clone classes from clone pairs.

Dup is another token based clone detection tool [4]. It finds identical clones as well as strictly parameterized clones, i.e., clones that differ only in the systematic substitution of one set of parameter values for another. These parameters can be identifiers, constants, field names of structures, and macro names but not keywords like *while* or *if*. The tool is text- and token-based and the granularity of clones is line-based where comments and white spaces are not considered. The algorithm used in this tool is based on a data structure called parameterized suffix tree [6], which is a generalization of a suffix tree. Dup is developed in C and Lex and only C code can be processed through Dup.

For finding maximal parameterized matching, Dup's lexical analyzer generates a string of one non-parameter symbol and zero or more parameter symbols for each line. The parameter and their positions are recorded in this parameterized string. This string is encoded in such a way that the first occurrence of each parameter is replaced by 0 and every later occurrence is replaced by the distance in the string since the previous occurrence. Non-parameter symbols are left unchanged. This tokenization scheme ensures the strict parameterization requirement of Dup. The clones detected by Dup cannot cross file boundaries but can cross function boundaries.

With the experience gained from the clone analysis of different software, it was observed that some short repeated segments of code turn up as multiple clone classes which are not very useful [22]. In Dup, these fragments of code had to be removed by hand. In CCFinder, the first 2 repetitions are reported and the rest are ignored. We have taken a different approach in Clone Miner. We do not report these short repetitions as clone classes, but rather we report them separately as repetitions.

There is also strong connection between this work and the work done previously on the design recovery and program understanding of large legacy systems for ease of maintenance and reuse [8][9]. Clones, especially clusters of cloned files as found by our approach, provide useful insights into the program structure for better understanding of the program. We expect that some of the structural clones may hint at important concepts behind a program. The contribution of clone detection towards program understanding is also discussed in [20].

9. CONCLUSION AND FUTURE WORK

In this paper, we described our strategy for detecting design-level similarity patterns, so-called structural clones, in programs. We apply token-based method to find simple clones first. For this, we use efficient suffix array based maximal repeats finding algorithm. Then, we apply data mining techniques to infer structural clones from simple clone data. With frequent closed itemset mining, we find the frequently occurring patterns of clones in different files. These patterns represent a clustering of files based on the similarity present among them. Finally, we perform a customized file clustering to extract the significant clusters, thereby filtering out the files that may contribute to a design level similarity pattern or structural clone.

We designed a tool prototype, called Clone Miner, which implements our clone detection strategy. Experiments have shown that our method and tool can successfully find useful design-level similarity patterns, candidates for unification with generic design solutions. Programs after such unification are easier to understand, modify and reuse.

To our best knowledge, the method described in this paper is the first attempt to detect design-level similarity patterns, beyond simple clones. It is also the first attempt to apply data mining techniques in the context of clone detection problem.

Currently, we have applied our method to detect basic forms of structural clones, namely similar classes, files and complex gapped clones. The flexibility in our method allows us to extend it for finding other, more complex types of similarity patterns by performing similar analysis at methods, classes, modules or components level. Extending our method and Clone Miner for such context and experimentation with recovery of higher-level design similarities in various application domains is the top priority for our on-going work.

Just like any design analysis activity, and also as any data mining application, clone detection is facilitated greatly with proper visualizations. Hence developing a graphical user interface for the Clone Miner is at the top of our priority list for future work. The visualizations will include browsing the source code detected as a clone instance, with the differences highlighted between the different instances of a clone class.

Another important aspect of extending the clone detection is to cater for multiple languages. Currently, only Java source can be analyzed, but extending the system to support other languages can be easily incorporated because of the flexible design of the system. The only language dependant part of the clone detector is the lexical analysis and some language specific heuristics to optimize clone detection. Once the token string is generated, the remaining steps would all be identical. A challenging feature would be to provide multiple language facility within the same system, i.e., to cater for software systems that are written in multiple languages like the web applications.

From the discussion on cluster mining, it is evident that real life clustering problems may have different and complex requirements as compared to the theoretical problems discussed in the literature. Sometimes we may require a unique solution tailored only for the given problem.

10. ACKNOWLEDGEMENTS

Many thanks are due to the following people for their invaluable support and guidance at different stages of the research and development: Bill Smyth, Simon Puglisi (for providing the implementation of the NERF algorithm and several useful suggestions), Katsuro Inoue, Toshihiro Kamiya, and Damith Chatura Rajapakse.

11. REFERENCES

[1] Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E. The enhanced suffix array and its applications to genome analysis. In *Proc. Workshop on Algorithms in Bioinformatics*, in Lecture Notes in Computer Science, vol. 2452, Springer-Verlag, Berlin, 2002, pp. 449–463.

[2] Abouelhoda , M. I., Ohlebusch, E., and Kurtz, S. Optimal Exact Strring Matching Based on Suffix Arrays. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval,* pages .31-43. September 11-13, 2002.

[3] ANTLR website at http:// www.antlr.org

[4] Basit, H. A., Rajapakse, D. C., and Jarzabek, S. Beyond Templates: a Study of Clones in the STL and Some General Implications. In *Proceedings of the 28th Intl. Conf. on Software Engineering (ICSE'05)(to appear).* 2005. Draft available at http://xvcl.comp.nus.edu.sg/xvcl_cases.php

[5] Baker, B. S. On finding duplication and near-duplication in large software systems. In *Proc. 2nd Working Conference on Reverse Engineering.* 1995, pages 86-95.

[6] Baker, B. S. Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance. *SIAM Journal of Computing*, October 1997.

[7] Baxter, I., Yahin, A., Moura, L., and Anna, M. S. Clone detection using abstract syntax trees. In *Proc. Intl. Conference on Software Maintenance* (ICSM '98), pp. 368-377.

[8] Biggerstaff, T.J. Design Recovery for Maintenance and Reuse. *Computer* 22(7), pp. 36-49, (July 1989).

[9] Buss, E., Mori, R. D., Gentleman, W., Henshaw, J., Johnson, H., Kontogiannis, K., Merlo, E., Muller, H., Paul, J. M. S., Prakash, A., Stanley, M., Tilley, S., Troster, J., and Wong, K., "Investigating reverse engineering technologies for the CAS program understanding project"**,** *IBM Systems Journal,* 33(3):477-500, 1994.

[10] *Case Study: eliminating redundant codes in the Buffer library*. At XVCL Website, http://xvcl.comp.nus.edu.sg/xvcl/buffer/index.htm

[11] Church, K. W. and Helfman, J. I. Dotplot: A program for exploring self-similarity in million of lines of text and code. *Journal of Computational and Graphical Statistics*, June 1993, 2(2):153-174.

[12] Davey, N., Barson, P., Field, S., Frank, R., and Tansley, D. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3-4): 219-236, 1995.

[13] Ducasse, S, Rieger, M., and Demeyer, S. A language independent approach for detecting duplicated code. In *Proc.*

Intl. Conference on Software Maintenance (ICSM '99), pp. 109-118.

[14] Fowler, M. *Analysis patterns: reusable object models*. Addison-Wesley, 1997

[15] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading Mass., Addison Wesley, 1995.

[16] Grahne, G., and Zhu, J., *Efficiently Using Prefix-trees in Mining Frequent Itemsets*. In *Proceeding of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Melbourne, FL, Nov 2003.

[17] Han, J., and Kamber, M. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco (2001).

[18] Jarzabek, S. and Shubiao, L. Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique. In *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2003, Helsinki, pp. 237-246.

[19] Java Technology at http://java.sun.com/

[20] Johnson, J. H., "Identifying redundancy in source code using fingerprints," Proc. of the 1993 Conf. of the Centre for Advanced Studies on Collaborative research: software engineering (CASCON '93), pp 171-183.

[21] Johnson, J. H. Substring Matching for Clone Detection and Change Tracking. In *Proc. Intl. Conference on Software Maintenance (ICSM '94)*, pages 120–126.

[22] Kamiya, T., Kusumoto, S, and Inoue, K. *CCFinder: A multi-linguistic token-based code clone detection system for large scale source code*. IEEE Trans. Software Engineering, vol. 28 no. 7, July 2002, pp. 654 – 670.

[23] Karkkainen, J., and Sanders, P. Simple linear work suffix array construction. In *Proc. 30th Internat. Colloq. Automata, Languages & Programming* (2003) 943-955.

[24] Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. Linear time longest common prefix computation in suffix arrays and its applications. *CPM 2001*, LNCS 2089.

[25] Kennedy, A., and Syme, D. Design and implementation of generics for the .NET Common Language Runtime. In Cindy Norris and James B. Fenwick, Jr., editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Languages Design and Implementation (PLDI-01)*, pages 1-12, New York, June 2001. ACM Press. Appears as volume 35, number 5 of SIGPLAN Notices.

[26] Kim, D.K., Sim, J.S., Park, H., and Park, K. Linear-time construction of suffix arrays. In *Proc. Fourteenth Annual Symp. Combinatorial Pattern Matching* (2003) 186-199.

[27] Ko, P., and Aluru, S. Space efficient linear time construction of suffix arrays. In *Proc. Fourteenth Annual Symp. Combinatorial Pattern Matching* (2003) 200-210.

[28] Kontogiannis, K.A., De Mori, R., Merlo, E., Galler, M., and Bernstein, M. Pattern Matching for Clone and Concept Detection. *J. Automated Software Eng.*, vol. 3, pp. 770-108, 1996.

[29] Komondoor, R., and Horwitz, S. Using slicing to identify duplication in source code. In *Proc. 8th International Symposium on Static Analysis*, 2001, pages 40-56.

[30] Krinke, J. Identifying Similar Code with Program Dependence Graphs. In *proceedings of the Eight Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2001, pp. 301-309.

[31] Larsson, N.J., and Sadakane, K. *Faster Suffix Sorting.* Technical Report LU–CS–TR:99–214, Lund University (1999) 20 pp.

[32] Manber, U., and Myers, G. Suffix arrays: a new method for on-line search. *SIAM Journal of Computing*, 22:935-48, 1993.

[33] Mayrand J., Leblanc C., and Merlo E. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. Intl. Conference on Software Maintenanc*e (ICSM '96), pp. 244-254.

[34] Morzy, T., Wojciechowski, M., and Zakrzewicz, M. Pattern-Oriented Hierarchical Clustering. *Advances in Databases and Information Systems*, *Proceedings Third East European Conference, ADBIS'99,* Maribor, Slovenia, 1999. Lecture Notes in Computer Science 1691, Springer Verlag, 1999.

[35] Morzy, T., Wojciechowski, M., and Zakrzewicz, M. Web Users Clustering. In *Proc. of the 15th International Symposium on Computer and Information Sciences*, Istanbul, Turkey, 2000, pages 374-382.

[36] Morzy, T., Wojciechowski, M., and Zakrzewicz, M. Scalable Hierarchical Clustering Method for Sequences of Categorical Values. In *Knowledge Discovery and Data Mining - PAKDD 2001.* In *Proceedings 5th Pacific-Asia Conference, Hong Kong, China.* April 16-18, 2001. Lecture Notes in Artificial Intelligence 2035, Springer Verlag, 2001.

[37] Parnas, D. Software aging. In *Proc. 16th International Conference on Software Engineering (ICSE 1994)*, pages 279 -287.

[38] Puglisi, S. J., Smyth, W. F., and Turpin, A. The performance of linear time suffix sorting algorithms. In *Proc. Data Compression Conference 2005*, to appear (2005).

[39] Ryan, A. P. J., Smyth, W. F., Turpin, A., and Xiaoyang Y. New suffix array algorithms -- linear but not fast? In *Proc. 15th Australasian Workshop on Combinatorial Algorithms*, Seok-Hee Hong (ed.) (2004) 148-156.

[40] Sadakane, K. A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation. In *Proc. IEEE Data Compression Conference* (1998) 129–138.

[41] Somerville, I. *Software Engineering*, Addison-Wesley Publishing Co., New York (1998).

[42] XVCL website at : http://xvcl.comp.nus.edu.sg/overview_brochure.php