Detection and Analysis of Clones in UML Class Models

Dhavleesh Rattan [(1),] Rajesh Bhatia [(2),] and Maninder Singh [(3)]

(1) Department of Computer Engineering. Punjabi University (India)
E-mail: dhavleesh@rediffmail.com

(2) Department of Computer Science and Engineering. PEC University of Technology (India)
E-mail: rbhatiapatiala@gmail.com

(3) Computer Science and Engineering Department. Thapar University (India)
E-mail: msingh@thapar.edu

ABSTRACT

It is quite frequent to copy and paste code fragments in software development. The copied source code is called a software clone and the activity is referred to as code cloning. The presence of code clones hamper maintenance and may lead to bug propagation. Now-a-days, model driven development has become a standard industry practice. Duplicate parts in models i.e. model clones pose similar challenges as in source code. This paper presents an approach to detect clones in Unified Modeling Language class models. The core of our technique is the construction of a labeled, ranked tree corresponding to the UML class model where attributes with their data types and methods with their signatures are represented as subtrees. By grouping and clustering of repeating subtrees, the tool is able to detect duplications in a UML class model at different levels of granularity i.e. complete class diagram, attributes with their data types and methods with their signatures across the model and cluster of such attributes/methods. We propose a new classification of model clones with the objective of detecting exact and meaningful clones. Empirical evaluation of the tool using open source reverse engineered and forward designed models show some interesting and relevant clones which provide useful insights into software modeling practice.

Keywords: Maintenance, Model clone, Model clone detection, UML Class models, Reverse Engineering.

1- INTRODUCTION AND MOTIVATION

Now-a-days modeling is playing a significant role in industries of different domains like automotive domain, web based applications and other complex systems [1]. Model driven software development using Unified Modeling Language (UML) improves the quality of product delivery and analysis of the product [2]. UML models bring in extra advantages of automatic code generation, early verification and validation [2]. Since modeling has been accepted as best practice, UML class models may also be analyzed for the presence of clones.

Clones in UML class models refer to the presence of duplicate parts, i.e. set of identical attributes, operations or both across different classes in the model. On the other side, code clone detection has been an active area of research for long [3]. Evidences suggest that the presence of code clones may lead to bug propagation and maintenance problems [4]. For example, if one fragment of code is changed, that change has to be carried out in all duplicate instances which may lead to missing or erroneous changes [5]. As UML models provide an abstract view of the system, thus detecting clones in models is equally important because similar challenges exist in the case of UML models [6], too.

An existing tool for clone detection in UML models is MQ_{lone} that presented model clone classification adapted from the work on source code clones [6]. Since UML modeling has got inherent object oriented features [7], thus our classification of model clones is inspired from these object oriented characteristics of UML class model where we view the model as a collection of logical entities defining data and behavior [8].

Primarily, the objective of our study is to detect model clones and present our classification and findings as observations of the characteristics of model clones. Our use case is to detect and analyze clones in forward designed UML class diagrams and open source reverse engineered class models. The detected exact and meaningful clones help in understanding the characteristics of UML models with regard to cloning. The major contributions of our approach are:

- Detection of model clones in UML class diagrams at different levels of granularity i.e. single attribute/operation*, set of attributes/operations* and recurring classes with their members*.
- Detection and classification of model clones as:
    - Type-1 : model clones due to standard modeling/coding practice
    - Type-2 : model clones by purpose
    - Type-3 : model clones due to design practices
- Carry out the empirical evaluation on reverse engineered open source system. Moreover, we are also considering forwarded designed model for evaluation to capture the essence of model driven development.

In this paper, the background of our work is mentioned in the next section. Some elaborative examples to understand our classification of model clones are present in section 3. Section 4 of the paper explains our approach of model clone detection. The empirical evaluation and results of the tool are included in Section 5. A brief survey related to work in model clone detection and comparison with existing work is done in section 6. Section 7 discusses the findings. Finally, section 8 concludes the work and briefly presents the scope for future work.

*For the sake of clarity, the member fields/attributes of a class with its data type will be referred to as a field/attribute and a method/operation of a class with its complete signature (return type and arguments) will be referred as a method/operation throughout the paper.

2- BACKGROUND

2-1 MODEL CLONE DETECTION

To know the current state of the art in model clone detection, we came across techniques in the literature to detect clones in Matlab/Simulink models [1],[9]-[11],[20]. Most of the approaches are graph based and clone detection is carried out by applying graph matching techniques. But in case of UML models significant differences emerge after we transform a model into a graph. Störrle [6] has stated this difference in his study that “UML models are not densely connected graphs of lightweight nodes, but rather loosely connected graphs of heavy nodes”. Therefore, a tree based approach is beneficial in exploring the heavy nodes of UML model, i.e. classes, its members (attributes/operations) and relationships among classes. In our approach, we construct a labeled ranked tree from the UML class model. The detail of our approach of model clone detection is present in section-4.

In a recent study by Saez et al. [13], a controlled experiment is carried out to compare the effectiveness of forward designed and reverse engineered models. The results of the study promoted the use of modeling as forward design models are easier to understand and maintain. Moving in the same direction, we evaluated our tool on forward designed and reversed engineered open-source UML class models.

2-2 REASONS FOR MODEL CLONES

Here we mention some of the reasons in brief that might lead to clones in UML models:

- Accidental Cloning
- Lack of restructuring and programmers’ limitation
- Reuse through copy and paste
- Language limitation

2-3 DEFINITIONS

We skip definitions of code clones, types and detection techniques in this paper. The interested reader may refer to detailed, recent survey by Rattan et al. [3]. Definitions necessary for evaluation and understanding of the proposed work are:

Definition 2.1 (Model Element)

A model element refers to a class, an attribute declaration (data type and name of the attribute) or a method signature (return type, name of the method and the data type of parameters) in a UML class model. It is abbreviated as ME throughout the paper. #ME refers to the total number of model elements extracted from the XMI [14] file of the UML class model.

Definition 2.2 (Model Clone)

A model clone is a pair (A,B) where A & B are identical sets of model elements present in the model. A set of model elements may contain a single model element or a group of model elements.

Definition 2.3 (Frequency)

It is the total number of occurrences of a particular model clone across the model. It is calculated for all the model clones.

Definition 2.4 (Clone Cluster)

A group of model elements present in a model clone is referred to as a clone cluster.

It is represented as triplet (uid, size, C) where

uid is the unique identifier of the cluster

size is the total number of model elements in the group

C is the frequency of clone cluster

This definition is purposely included for classifying model clones as type-2 and type-3 as discussed in detail in section-5, where a detailed evaluation is given for different subject systems.

Definition 2.5 (Coverage)

It defines the number of duplicate model elements vs. total model elements (#ME). In other words, it determines the probability that a random model element is part of a clone. E.g. in next section in Fig. 1 on library management system, there are 24 model elements. Out of these, 4 are cloned ME. So the coverage % is 16.6%. It specifies the extent of duplication across classes in a UML class diagram.

Clone coverage has been successfully used in evolutionary analysis of software systems like Linux kernel [12]. By the same token, we believe that clone coverage be applied in the evolutionary study of model clones as well.

Göde et al. [15] has mentioned clone coverage as an important cloning metric. The study highlight minimum clone length, normalization, structure and design of the subject system as key factors affecting clone coverage. In our paper, we have taken the minimum clone size to be 1 (a single model element) and it is done uniformly for all the subject systems to make the results usable. We understand that when the minimum clone length is increased, clone coverage may decrease.

3- MODEL CLONE DETECTION BY EXAMPLE

The objective of our technique is to automatically detect exact and meaningful clones. Thus we propose following categories with an explanation using different examples.

Consider the sample UML class model of a library management system shown in Fig.1. From the domain perspective, this model appears to be simple; however, in the context of model clone detection, it will serve our purposes well.

Some fields/operations tend to repeat a lot in UML class model and in code. As shown in Fig. 1, we notice that field *id: Integer* is present in most of the classes. It is a modeling practice to uniquely identify an instance of an entity. We named such clones to be **type-1-Model clones due to standard modeling/coding practice**. We noticed such repetitions in reverse engineered class diagrams, too. One of them is *serialVersionUID*, which is used as a version control in Serializable class. The fact is that the serialization runtime associates with each serializable class a version number which is used during deserialization to verify the sender and receiver. If *serialVersionUID* is not explicitly declared by a serializable class, then the runtime will calculate a default *serialVersionUID* for that class.

Another example is *readObject* and *writeObject* which are used in Java's serialization to read and write byte stream in physical location. Most of these technical details surface in reverse engineered models and are unlikely to appear in realistic models. It may seem irrelevant in the first observance, but we intend to bring them into notice as well. Type-1 clones most of the time consists of a single attribute/operation. Thus the minimum clone length is set to 1 for identifying such repeating field/method. Surely, the presence of these fields in classes suggests application of programming/modeling practice.

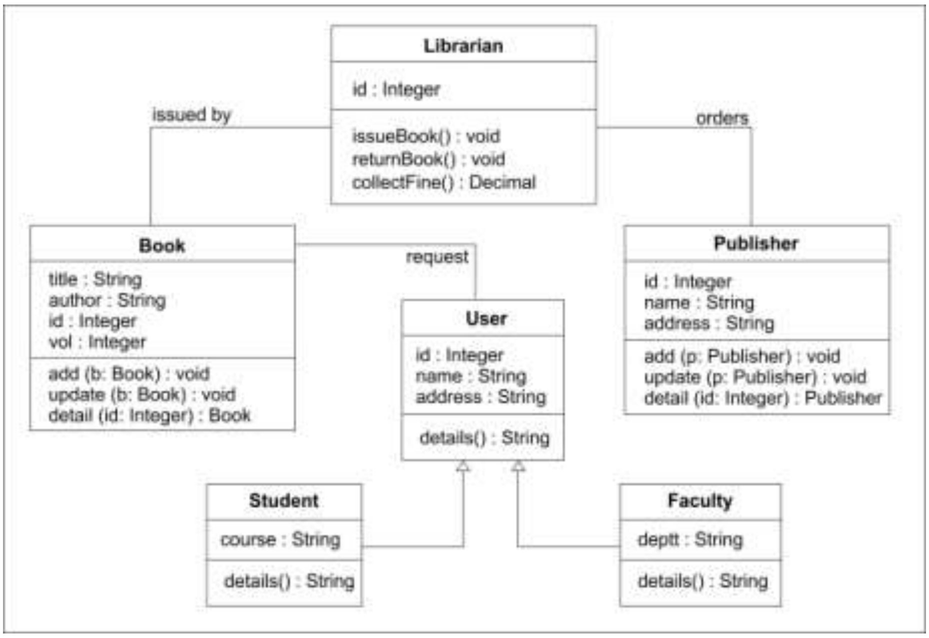


Figure 1 Class diagram for library management system.

Fig. 1 also shows a generalization relationship. The user generalization is shown as *Student* class and *Faculty* class to be the subclasses of the *User* class. Both the subclasses override an operation named *details (): String* from the parent class. The *details (): String* operation must be implemented for each kind of user. This model clone is reported to occur in parent and all the subclasses because of the nature of the relationship. In other examples, there are several reasons why one may wish to override a feature. These types of clones are categorized as **Type-2-Model clones by purpose**. In a nutshell, overriding is an important and useful language feature in object oriented programming.

Interfaces are an integral part of UML class diagrams. An interface is a class like construct which shows a collection of operations that specify a service to a class. In a UML class model, realization relationship is used to show an abstraction and its implementation. Fig. 2 shows an interface *ChoiceBlock* with two abstract methods *setDefault (choice: Choice)* and *getChoice (): Choice*. *ChoiceBlock* aims to specify the behavior of classes *PopUpMenu* and *RadioButtonArray* using realization relationship. Both of these classes implement the abstract methods specified in the interface. A number of programming languages support the concept of interfaces. E.g. in Java programming language interface keyword is used to specify an interface. This repetition of abstract operations in the interface and its implemented classes is also categorized as **Type-2-Model clones by purpose**.

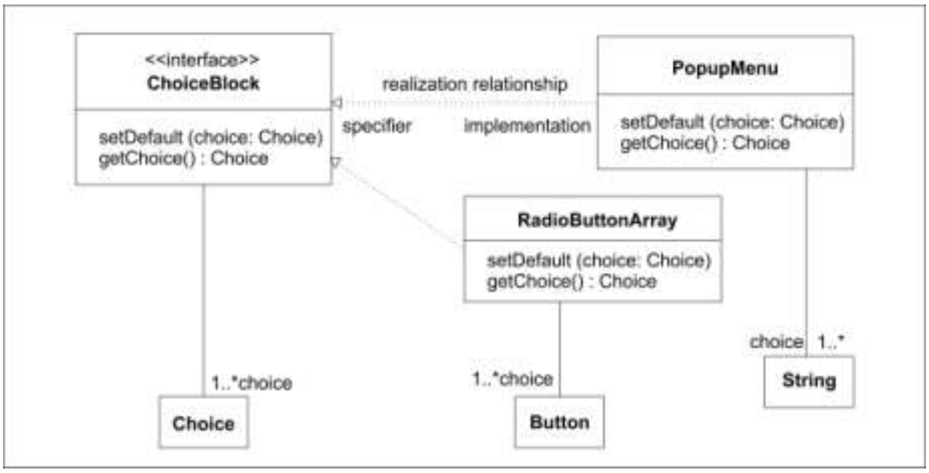


Figure 2 Realization relationship [16].

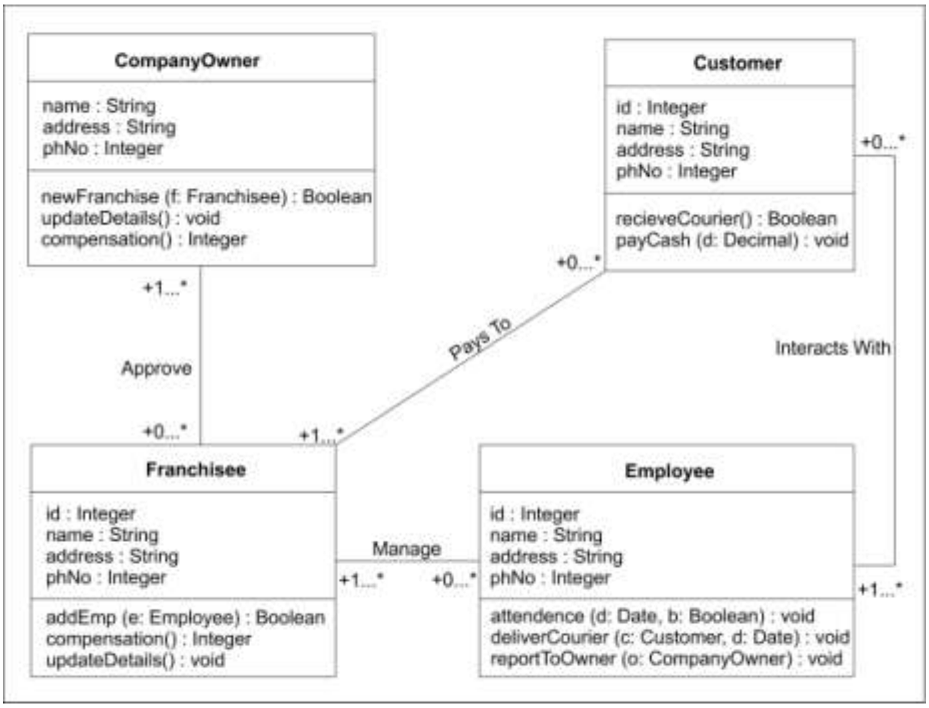


Figure 3 A class diagram for courier mangement system.

In Fig. 3, a class diagram of the Courier Management System is shown. Among all the classes viz. *CompanyOwner*, *Customer*, *Franchisee* and *Employee*, we see a repetition of a group of fields and methods. In the figure, there is a large cluster of size 5 i.e. *address: String, compensation (): Integer, name: String, phNo: Integer, updateDetails (): void* which is present in 2 classes namely, *CompanyOwner* and *Franchisee*. The second cluster of size 4 is *id: Integer, name: String, address: String, phNo: Integer* which is present in 3 classes namely, *CompanyOwner, Customer* and *Employee*. The last cluster of size 3, viz. *name: String, address: String* and *ph No: Integer* is present in all the classes. This type of repetition in different classes as reported by the proposed technique is categorized as **Type-3-Model clones due to design practices**. Table 1 shows all these clusters together with the type of clone. This may be the result of unfinished design or due to any other possible reason. Thus, these clones may subject to further improvement in the design w. r. t. maintainability and extensibility.

Table 1 Analysis of courier management system

Number of different model elements		**Total = 17; Classes= 4; Methods= 9; Attributes= 4**			
Sr. No.	**Name of attribute/method**	**Freq**	**Classes**		
1.	*compensation(): Integer, updateDetail(): void*	2	*CompanyOwner, Franchisee*		
2.	*id: Integer*	3	*Customer, Employee, Franchisee*		
3.	*Address: String, name: String, phNo: Integer*	4	*CompanyOwner, Customer, Employee, Franchisee*		
Coverage %		35% (6/17)			
Clone Cluster		**UID**	**Size**	**C (Classes)**	**Type of clone**
		1	3	4	Type-3
		2	4	3	Type-3
		3	5	2	Type-3

Table 1 shows the results after applying the proposed technique on the UML class model shown in fig. 3. In total we get 17 model elements comprising of 4 classes, 4 attributes and 9 methods. The list of model elements with their frequency and list of classes in which they appear are shown in table 1. The table lists the coverage % of the model. This is the number of cloned model elements vs. total number of elements. Among the total 17 model elements, 4 fields and 2 methods are repeating. It gives 35% coverage.

4- PROPOSED APPROACH FOR DETECTING MODEL CLONES

In this section, we formalize our approach of model clone detection that consists of three broad steps. Firstly, we export the UML class model to XMI file and parse the XMI file to construct a labeled, ranked tree. In the second step, we apply the algorithm to detect model clones in the constructed tree. In the final step we classify the detected model clones into different categories. Fig. 4 shows the block diagram of the proposed technique.

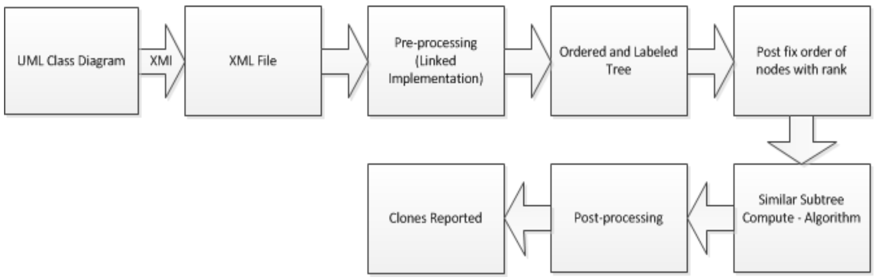

UML Class Diagram
XMI
XML File
Pre-processing (Linked Implementation)
Ordered and Labeled Tree
Post fix order of nodes with rank
Similar Subtree Compute - Algorithm
Post-processing
Clones Reported

Figure 4 Block diagram of the approach.

4-1 MODELING AND PREPROCESSING

Modeling:

We use MagicDraw Enterprise 16.6 CASE tool for creating forward designed UML class model and for reverse engineering open source subject system. The model is then exported to an XMI file using XMI export facility of the CASE tool. The exported XMI file is input into our tool to parse and extract information related to our clone detection process.

Preprocessing:

The exported XMI file contains a lot of tool specific information along with the class diagram/model data. Thus, the parsing of XMI file is a major step to extract data of interest i.e. model elements. The XMI elements/nodes are then realized into a tree structure equivalent to the model keeping in mind the constructs of language like nesting of packages, declaration of classes, fully qualified definition of attributes and operations (model elements). Generally, when the UML class model is created, the members of the class may be added in no particular order. So, during parsing of XMI file, these model elements are fetched and stored in lexicographic order in the tree.

The algorithm to construct the tree and related definitions are as follows.

Definition 4.1.1 (Element)

It is a common term that refers to a package or class or child class or field (attribute) or method (operation).

Definition 4.1.2 (PackageNode)

It stores information about a package element, its sub packages and classes.

Definition 4.1.3 (ClassNode)

It stores information about a class element, its inner-classes, fields and methods.

Definition 4.1.4 (ModelTree)

It is the tree constructed from model elements.

Definition 4.1.5 (Rank)

It is the out-degree of a node of the tree.

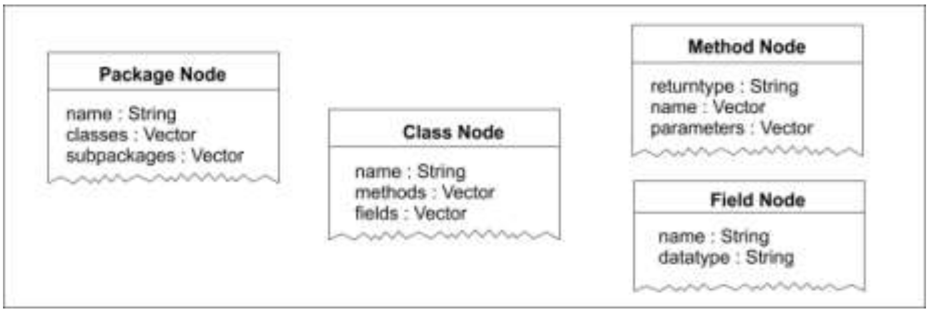


Figure 5 Different nodes of algorithm 1 as implemented.

Fig. 5 shows different nodes defined above. These are used in Algorithm1 for storing data extracted from XMI file of the model for pre-processing.

Following is the algorithm applied to construct the ModelTree:

Algorithm 1: Tree Construction Algorithm

Input: An XMI file of UML class model exported by MagicDraw

Output: The post-fix traversal of the ModelTree with rank

find: The root package of UML model and store it in ModelTree

1) Read all the children of the root package node.

2) If the child is a package then store it in the ModelTree under its parent PackageNode and read all the children of this package recursively till all the packages are read.

3) If the child is a class then store it as ClassNode in the ModelTree under its parent PackageNode

4) Read all the children of the ClassNode recursively

4.1) If the child is a field, then read its label and data type then add to the class's fields.

4.2) Else if the child is a method, then read its label, return type and paramec xc 65gter types and add to the class's methods.

4.3) Else if the child is an inner-class then store it in the ModelTree under its parent ClassNode from step 3.

5) Traverse the constructed ModelTree in postfix order and store it in a list.

4-2 DETECTING CLONES

In the preprocessing phase, we constructed the ModelTree from the UML class model.

Fig. 6 shows the tree representation of UML class diagram shown in fig. 3 (Courier Management System). An attribute model element consists of data type and the name. This is shown as a subtree consisting of two nodes, i.e. a data type node with the attribute name as its child node. Various attribute model elements are depicted in red color in fig. 6. Similarly, a method model element is also represented as a subtree with its return type, name and arguments located at different levels of the same subtree. Various method model elements are shown in blue color in the above figure. Therefore, detecting repeats of these subtrees in the model tree is essential to find model clones. To achieve this, we are using the approach by Christou et al. [17] to compute subtree repetitions. The technique is based on accepting the postfix string representation of the tree and computing subtree repeats with varying sizes in a bottom-up manner. The algorithm has got linear time and space complexity.

The tree diagram in fig. 6 also shows the postfix string representation of the model tree on the right hand side. The rank (out-degree of a node) in the tree is shown on the top of the respective node.

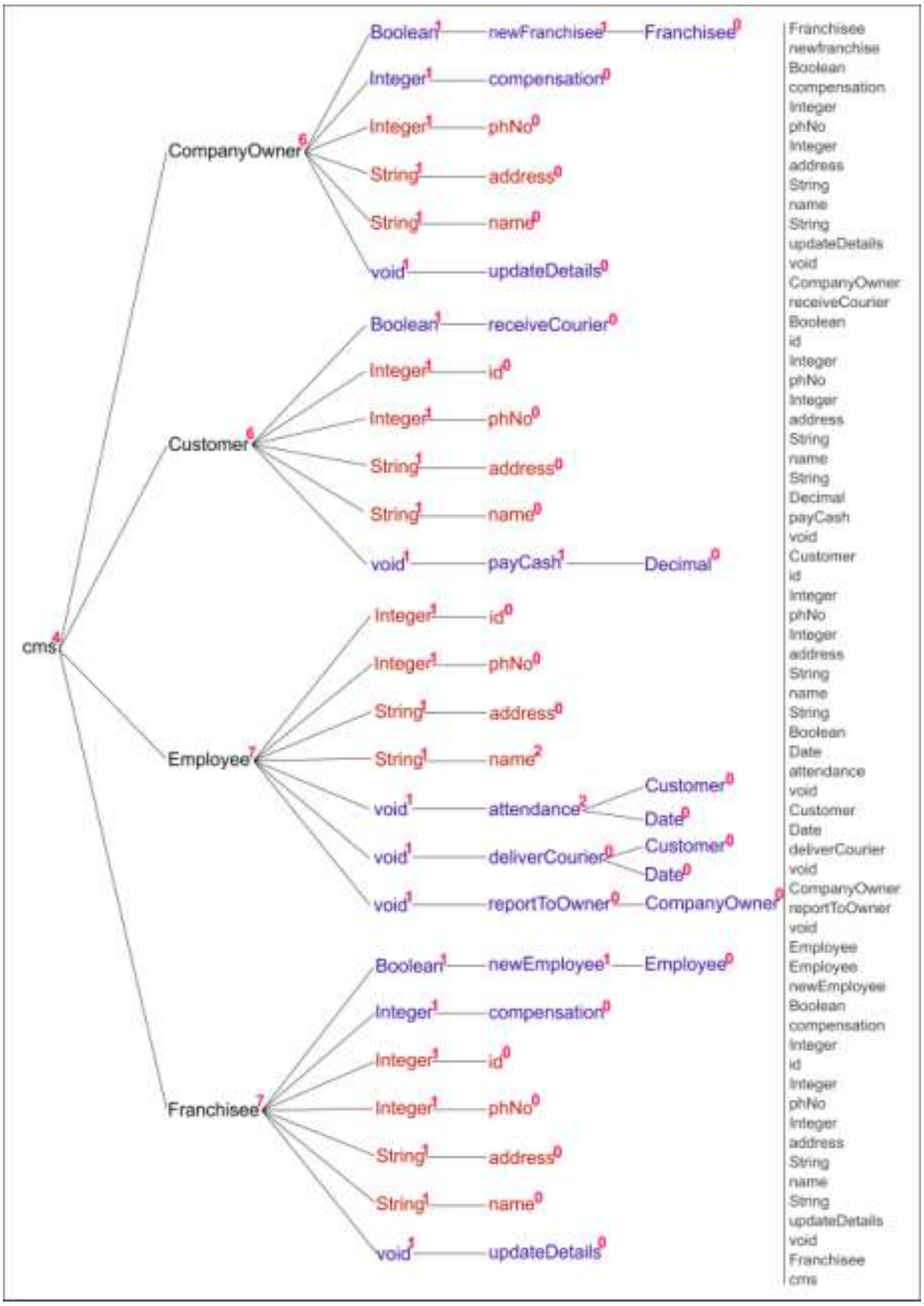
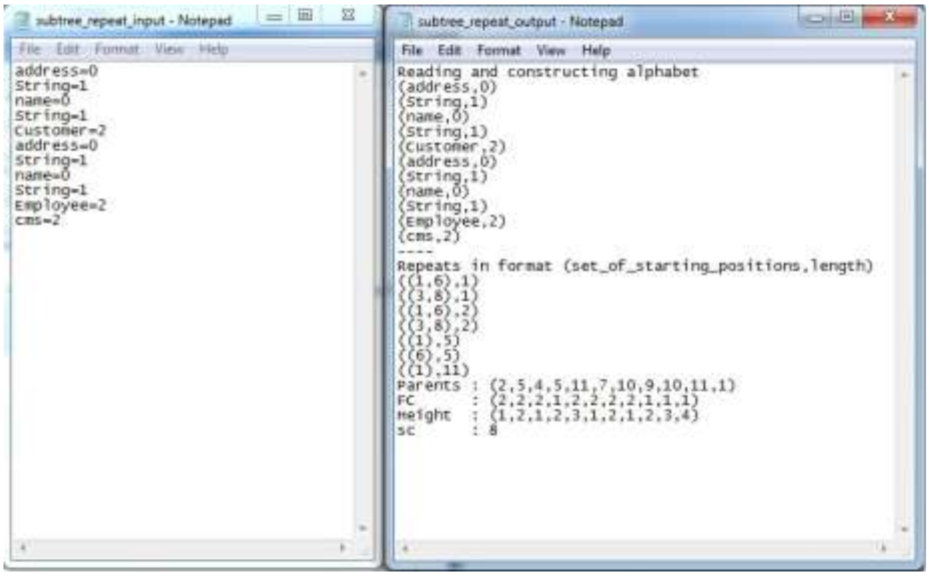


Figure 6 Tree representation of the UML class model.

4-3 POST PROCESSING AND CLUSTERING CLONES

The output of previous phase reports identical subtrees in the form: set of starting positions, length in corresponding to the input postfix string representation. We need to apply extensive post-processing techniques to make the results useful in a way we require to present the classification and granularity of our approach. To achieve this, the tool maps the output of algorithm to the model tree (constructed in pre-processing phase) for retrieving the results.


subtree_repeat_input - Notepad
File Edit Format View Help
subtree_repeat_output - Notepad
File Edit Format View Help

```
address=0
String=1
name=0
String=1
Customer=2
address=0
String=1
name=0
String=1
Employee=2
cms=2
```

```
Reading and constructing alphabet
(address,0)
(String,1)
(name,0)
(String,1)
(Customer,2)
(address,0)
(String,1)
(name,0)
(String,1)
(Employee,2)
(cms,2)
----
Repeats in format (set_of_starting_positions,length)
({1,6},1)
({3,8},1)
({1,6},2)
({3,8},2)
({1},5)
({6},5)
({1},11)
Parents : (2,5,4,5,11,7,10,9,10,11,1)
FC      : (2,2,2,1,2,2,2,2,1,1,1)
Height  : (1,2,1,2,3,1,2,1,2,3,4)
SC      : 8
```

Figure 7 The input and output of the subtree repeat algorithm [17] for a sample class model.

We intend to demonstrate this with an example. A small class diagram having two classes with a pair of repeating attributes is created. Fig. 7 shows the actual output of the algorithm [17] in the form of sets of starting positions of repeating subtrees and their lengths. As this output cannot be directly applied to our approach, so the post-processing phase becomes an integral part of the whole process.

Following is the procedure to calculate frequency of cloned ME, grouping and clustering.

Note: ***OPL*** refers to ordered postfix list. ***l*** refers to label which is either a field or method. ***Ci*** indicates classes in ModelTree. ***CRM*** refers to class relationship matrix. A matrix element with non-zero value signifies relationship between classes or interfaces. Each relationship type is denoted with a unique identifier.

A) Frequency:

Firstly, the tool applies following algorithm to distinctly report model clones with frequency 2 and model clones with frequency more than 2. The latter set of model clones is inspected to state type-1 clones.

Algorithm

```
1. For each repeating l ∈ OPL
        1.1) [RepeatsList] ← l
2. For each l ∈ [RepeatsList]
        2.1) Count the occurrences, o of l in OPL
        2.2) [NewList] ← (l, o)
        2.3) l is categorized as field or method in [NewList]
```

B) Grouping:

This step is the prerequisite for clustering algorithm.

Algorithm

```
1. For each l ∈ [RepeatsList] && Ci ∈ ModelTree
        1.1) If l ∈ Ci
                1.1.1) [GroupingList] ← ( l, C1, C2, ..,Cn)
                1.1.2) l is categorized as field or method
2. For each Ci ∈ ModelTree && (l ∈ [RepeatsList] && l ∈ Ci)
        2.1) [RepeatsinClass] ← l
        2.2) l is categorized as field/method
```

[RepeatsinClass] stores the repeating fields/methods for each class, with their count.

C) Clustering:

To classify and report the clones as type -2 and type-3 model clones, the tool

1) Cluster the duplicate model elements

2) Find occurrence of such cluster across classes

3) Determine the relationship between those classes

Algorithm

```
1. For each Ci ∈ ModelTree
        1.1) For each I ∈ Ci && I ∈ [RepeatsinList]
        1.2) Store repeating I for corresponding Ci
[Traverse the RepeatsInClass and get a list of repeating members for that Class]
2. For each Ci ∈ ModelTree
        2.1) Compare [RepeatsInClass] for Ci in Step-1 to the Ci in this step (Step-2)
3. [Cluster] ← Ci
[Clusters of repetitive members (Fields/Methods) and their presence in Classes through-out the model]
4. Determine the nature of relationship from CRM among the classes from step-3.
```

4-4 EXPERIMENTAL SETUP

The proposed technique has been implemented in Java and runs under Windows 7. It is capable of detecting model clones in any object oriented class model.

Table 2 Hardware configuration and software tools

Processor	Intel (R) Core (TM) i3-3110M CPU@ 2.40GHz
Installed Memory (RAM)	4.00 GB
System Type	32-bit Operating System
Windows Edition	Windows 7 Professional N Service Pack 1
Integrated Development Environment	• NetBeans IDE • Bloodshed Dev-C++

Software Development Kit	Java Development Kit 6
Software Modeling Tool	MagicDraw

The tool receives XMI file of the class model as input and reports the clones as per the defined classification with the count. The tool uses the given algorithm by Christou et al. [17] to compute similar subtrees which has linear time and space complexity. Table 2 lists the hardware configuration and software tools needed in the implementation.

4-5 PERFORMANCE ANALYSIS

Our tool can detect the model clones as proposed in the technique very efficiently. Since our tool is a multiphase detection approach, we tried to estimate the computational time of each phase using an average of five iterations for the same input. The execution time is reported in milliseconds. It takes less than 1 minute to detect model clones in *eclipse-ant*, *eclipse-jdtcore* and *netbeans-javadoc* with more than 290 model elements.
Table 3 shows the actual measurement of time when the tool is executed on a PC with the configuration as given in table 2.

Table 3 Execution time

	eclipse-ant	**eclipse-jdtcore**	**netbeans-javadoc**
	Average of 5 runs	*Average of 5 runs*	*Average of 5 runs*
Preprocessing and Transform	209	322	217
Computing Subtree Repeats	15	11	13
Analysis	18	30	16
Grouping and Reporting	81	30	63
	323 ms	**393 ms**	**309 ms**

The UML class model is created using the MagicDraw modeling tool. Then the class model is exported to XMI file which is given as input to the first phase of the tool.
Preprocessing and Transform Input: In this phase, the XMI file of the given UML class model is parsed and the relevant information is extracted from the XMI file which includes parameters like the name of the class, name of the attribute with their data type, methods with their return type and its arguments, etc. These parameters are then realized into a tree structure i.e. ModelTree equivalent to the model keeping in mind the constructs of language.
Computing Subtree Repeats: We used the algorithm [17] to computer similar subtrees repeating in the model tree made up of class model. The technique is based on accepting the postfix string representation of the tree and computing subtree repeats with varying sizes in a bottom-up manner. The algorithm

has got linear time and space complexity. It reports identical subtrees in the form: set of starting positions, length in corresponding to the input postfix string representation.

Analysis: In this phase, the tool reports model clones with frequency 2 and model clones with a frequency more than 2. The latter set of model clones is inspected to state type-1 clones.

Grouping and Reporting: In this phase, the tool reports different field and methods repeating across classes. It gives the corresponding classes too. Finally, it clusters repeating field and methods which are present in more than one class together with the relationship between those classes.

The tool reports the clones in a way which require subjective assessment from a developer's viewpoint.

5- EMPIRICAL EVALUATION

Empirical evaluation of the proposed technique is carried out on forward designed and reverse engineered UML class models. One of the UML models is forward designed i.e. created during the normal development life cycle. Such a model helps to check the practical relevance of the proposed approach. Moreover, no standard repository of UML models recognized by the modeling community is available; therefore we chose class diagram created by reverse engineering open source subject systems. It will provide a platform for the research community to compare and verify the results in the future. We chose the subject system from Bellon's experiment [18] for empirical evaluation as these systems are well known in code clone detection community. The earlier work on model clone detection for UML models, i.e. MQ_{lone} [6] was evaluated on different models created by 16 Master's students. The models are not publicly available. Our subject systems are open source, diverse, industrial size and widely used for development. It may help in understanding the nature of heterogeneous subject systems with respect to cloning. On the other hand, MQ_{lone} was evaluated on a set of homogeneous subject systems.

Subject Systems: The forward designed model is a class model for Proxy Server created using MagicDraw Enterprise 16.6 tool. Other subject systems of our study are reverse engineered using the same tool. The characteristics of the system are listed in Table 4.

Table 4 Subject system

Subject System	**Language**	**Program Size (SLOC)**	**#ME**	**XMI File Size (in KB)**
eclipse-ant	Java	35K	292	947
eclipse-jdtcore	Java	148K	174	1445
netbeans-javadoc	Java	19K	267	798

Table 5 reports the number of cloned model elements. Clones (f=2) shows the total count of model clones with frequency 2. Clones (f>2) states model clones repeating more than two times.

Table 5 Clones in subject systems

Subject System		#ME	Cloned ME	Clones (f=2)	Clones (f>2)
eclipse-ant	Classes	30	00	00	00
	Attributes	106	17	13	04
	Methods	156	28	15	13
eclipse-jdtcore	Classes	08	00	00	00
	Attributes	85	00	00	00
	Methods	81	00	00	00
netbeans-javadoc	Classes	27	00	00	00
	Attributes	66	08	05	03
	Methods	174	21	17	04
Proxy-Server	Classes	10	00	00	00
	Attributes	33	2	2	0
	Methods	62	3	00	3

The results of the evaluation are presented for the subject systems in a consistent manner, reporting the model clones with high frequency. Next, clusters of attributes/methods present in different classes are reported. Evaluation is based on following three types of clones:

1. Type - 1: Model clones due to standard modeling/coding practice
2. Type - 2: Model clones by purpose
3. Type - 3: Model clones due to design practices

Above categories of clones are explained in detail with examples in Sec 3 – model clone detection by example. At last, for every class model coverage is mentioned to know the percentage of cloned model elements out of total model elements. The evaluation of our work provides observations of the characteristics of model clones which may help in gaining some insights from an implementation point of view. In the earlier study [6], the clones are categorized as Type A (exact model clone), Type B (modified model clone) and Type C (renamed model clone) adapted from code clone classification (Type 1/ Type 2/ Type 3). The objective of his approach is to detect clones with varying degree of changes, whereas our technique is aimed at identifying clones which are actually relevant for practitioners.

5-1 CLONES IN ECLIPSE-ANT

Eclipse-ant is a freely available Java based build tool. The total number of model elements extracted from the XMI file is 292. Table 5 shows that out of total 30 classes, no class is repeating. There are about 17 attribute clones out of 106 attributes and 28 method clones out of 156 repeating across the complete UML class model. Fig. 8 shows a bar chart showing the repetition of attributes and methods across the model. There is one field *target: Target* and one method *startElement (String, attributeList): void* appearing seven times in the complete class model. The field *target: Target* is present in 7 classes viz. *BuildEvent, BuildSmallEvent, DataTypeHandler, NestedElementHandler, TargetHandler, Task and TaskHandler*. Similarly the method *startElement (String, AttributeList): void* is found in 7 classes namely, *AbstractHandler, DataTypeHandler, NestedElementHandler, ProjectHandler, RootHandler, TargetHandler and TaskHandler*. Except the class *RootHandler*, remaining 6 classes are in generalization hierarchy with root class *AbstractHandler*. Above such set of model clones detected by the tool need to be inspected to categorize them as Type-1 clones if these are the result of standard modeling/coding practice.

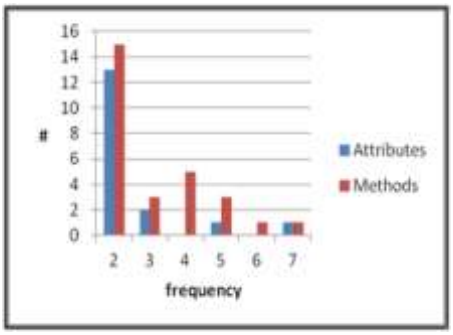


Figure 8 Eclipse-ant.

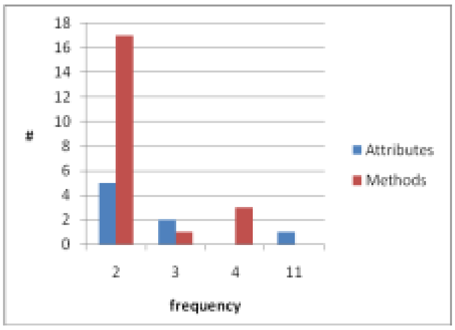


Figure 9 Netbeans-javadoc.

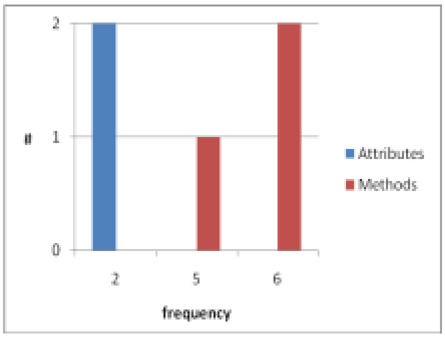


Figure 10 proxy-server.

Table 6 Type-2 and Type-3 clones in eclipse-ant

Type of Clone	**Type-2 Clone**		**Type-3 Clone**
Nature of relationship across classes	Inheritance	Realization	No relationship
No. of clones	3	2	13

5-1-1 Clone Clusters

The tool reported 18 clusters in eclipse-ant and categorized these clusters, as Type -2 and Type -3 clones on the basis of the relationship between the classes in which these clusters are present. This is shown in Table 6.

Type-2 clones: Model clones by purpose

Among type-2 clones, the tool reported 3 clusters repeating across generalization hierarchy in the class diagram. The first cluster consists of 2 methods viz. *characters (char [], int, int): void, startElement (String, AttributeList): void* present in *AbstractHandler, DataTypeHandler, NestedElementHandler* and *TaskHandler* classes. In another case, there is a multilevel inheritance across *ProjectComponent, Task* and *UnknownElement* classes. The cluster consisting of three methods, i.e. *execute (): void*, *getTaskName (): String*, *maybeConfigure (): void* is present in *Task* as well as *UnknownElement* class. Another cluster made up of 3 methods, namely, *characters (char [], int, int): void, finished (): void, startElement (String, AttributeList): void* is reported in *AbstractHandler* and *TaskHandler* classes.

During analysis, we came across a couple of instances where a set of concrete classes realize the same interface. In one such case, a cluster of 7 methods viz. *buildFinished (BuildEvent): void, buildStarted (BuildEvent): void, messageLogged (BuildEvent): void, targetFinished (BuildEvent): void,targetStarted (BuildEvent): void, taskFinished (BuildEvent): void, taskStarted (BuildEvent): void* is repeating in *AntClassLoader, DefaultLogger, IntrospectionHelper* and *XmlLogger* classes. These 4 classes realize an interface named *BuildListener* made up of just mentioned 7 methods.

Type -3 clones: Model clones due to design practices

The tool reported 13 clusters of type-3 clones repeating across classes that do not have any relationship among them. In the first case, 1 method, i.e. *log (String, int): void* and 1 field, i.e. *project: Project* is found in 2 classes *AntClassLoader* and *ProjectComponent*. There is no relationship between these classes. Similarly, 2 methods *getLocation(): Location* and *setLocation (Location): void* and 1 field *location: Location* is present in *BuildException* and *Task* classes. Another large cluster of same type, consisting of 5 fields viz. *exception: Throwable, message: String, priority: int, target: Target, task: Task* and 7 methods viz. *getException (): Throwable, getMessage (): String, getPriority (): int, getTarget (): Target, getTask (): Task, setException (Throwable): void, setMessage (String, int): void* is found in *BuildEvent* and *BuildSmallEvent* classes. In total there are 7 such instances, but we have listed only some of them here.

In an interesting case, there are five classes viz. *DataTypeHandler, NestedElementHandler, ProjectHandler, TargetHandler*, and *TaskHandler* which are the child classes of *AbstractHandler* parent class in the UML class model for eclipse-ant. Different clusters of fields/methods repeat across these five classes, but not in the parent class. E.g. one of the clusters consisting of 2 fields *target: Target, wrapper: RuntimeConfigurable* and three methods viz. *characters (char [], int, int): void, init (String, AttributeList): void, startElement (String, AttributeList): void* is present in *DataTypeHandler* and *TaskHandler* class. Another similar cluster comprising of two methods, i.e. *init (String, AttributeList): void, startElement (String, AttributeList): void* and one field *target: Target* present in *DataTypeHandler, NestedElementHandler, TargetHandler* and *TaskHandler* classes. Similarly, among the same five subclasses, a cluster of 2 methods, i.e. *init (String, AttributeList): void* and *startElement (String, AttributeList): void* is present.

Another similar cluster consists of 1 field and 3 methods existing in *DataTypeHandler, NestedElementHandler* and *TaskHandler* classes. In total we traced 6 such instances in which the duplication is prevalent across subclasses of the same parent class.

5-1-2 Coverage

In total, of 292 model elements (# ME) in eclipse-ant, we detected that 45 out of it are clones. These may be attributes/operations in a class. So clone coverage is total cloned elements vs. total elements, i.e. 15.4 %.

5-2 CLONES IN NETBEANS-JAVADOC AND ECLIPSE-JDTCORE

The technique has been applied to two systems, namely, *netbeans-javadoc* and *eclipse-jdtcore* with 267 and 174 model elements. Primarily, we want to know the degree of cloning in the form of individual attributes/methods and clusters in UML class models. *Netbeans-javadoc* has 29 cloned *ME* consisting of 8 attributes and 21 methods. *Eclipse-jdtcore* has no attribute/method, repeating out of 174 model elements as shown in Table 5. Fig. 9 shows the frequency of cloned model elements for *netbeans-javadoc*. 75 % of model elements in *netbeans-javadoc* are repeating two times. There is one attribute *serialversionUID* which is repeated across 11 classes. Individual attributes/operations have relevance during forward engineering as we generate code from mode as well as during reverse engineering as we generate model from the code. Though these types of clones seem small in the first observance, these signify standard modeling/coding practice, thus classified as Type-1 clones.

Clone clusters

In total, 6 clone clusters are detected. As shown in Table 7, all these clusters are present across classes where there is no relationship between the classes. We list some of them here. One clone cluster has 3 methods, namely, *getAdditionalBeanInfo (): BeanInfo*, *getBeanDescriptor (): BeanDescriptor*, *getIcon (int): Image* is present in 4 classes viz.

ExternalJavadocExecutorBeanInfo, *GlobalLocalFileSystemBeanInfo*, *JavadocTypeBeanInfo* and *StdDocletTypeBeanInfo*. Another large cluster comprising of 2 fields, namely, *docletS: StdDocletType* and *javadocS: ExternalJavadocSettingsService* and 5 methods *getDestinationDirectory (): String*, *isStyle1_1 (): boolean*, *loadChoosenSetting (): void*, *setBooleanOption (Boolean, String, list): void*, *setStringOption (String, String, List): void* is repeating in classes *ExternalOptionListProducer* and *OptionListProducer.*

Table 7 Type-2 and Type-3 clones in netbeans-javadoc

Type of Clone	**Type-2 Clone**		**Type-3 Clone**
Nature of relationship across classes	Inheritance	Realization	No relationship
No. of clones	00	00	06

In two particular cases, we came across four classes, namely, *ExternalJavadocExecutorBeanInfo*, *GlobalLocalFileSystemBeanInfo*, *JavadocTypeBeanInfo*, *StdDocletTypeBeanInfo* where all the methods *getAdditionalBeanInfo (): BeanInfo []*, *getBeanDescriptor (): BeanDescriptor*, *getIcon (int): Image* repeat. In another example, a group of 1 field, viz., *bundle: ResourceBundle* and 1 method, viz., *getBundledString (String): String* is present in 2 classes i.e. *CommonUtils* and *ResourceUtils*. We observed upon analysis of the results, 4 classes *ExternalJavadocExecutorBeanInfo*, *GlobalLocalFileSystemBeanInfo*, *JavadocTypeBeanInfo* and *StdDocletTypeBeanInfo* are exactly same. Each class is made up of same 3 methods as mentioned earlier. Another case is *CommonUtils* and *ResourceUtils* with similar contents.

Coverage

We get 11 % coverage in case of *netbeans-javadoc*. We detected 29 cloned *ME* in the total of 267 model elements.

5-3 CLONES IN PROXY-SERVER CLASS DIAGRAM

The proxy server class diagram is the forward designed class model. This class diagram is made up of 10 classes and 1 interface. In total we have 33 fields and 62 methods in the model. Fig. 10 shows a bar chart showing the repetition of attributes and methods across the model. To analyze the type-1 clones, we came across 2 fields viz. *SerialVersionUID: long* and *LOGGER: logger* which is repeating in 2 classes. As per our classification, these fields are the result of standard modeling practice.

5-3-1 Clone Clusters

In this class diagram, the tool detected 2 clusters. First cluster is made up of three methods, namely *onRequest (HttpServletRequest, HttpServletResponse, URL): void, onRemoteResponse (httpMethod): void* *and onFinish (): void* repeating in 5 classes *MimeTypeChecker, HostChecker, MethodsChecker, HostNameChecker, RequestTypeChecker* and 1 interface named *ProxyCallBack*. This is the result of classes implementing the same interfaces, thus having similar set of methods. These types of clones show behaviors in model emerged from the careful application of useful design paradigms and classified as a type-2 clones.

As shown in Table 8, there is another cluster made up of 2 methods *onRemoteResponse (httpMethod): void* and *onFinish (): void* which is present in the class *HTTPProxy* in addition to all the five classes mentioned earlier in the first cluster. Since *HTTPProxy* class has no relationship with any other class, thus categorized as type-3 clone.

Table 8 Type-2 and Type-3 clones in proxy-server

Type of Clone	**Type-2 Clone**		**Type-3 Clone**
Nature of relationship across classes	Inheritance	Realization	No relationship
No. of clones	00	01	01

5-2-2 Coverage

We found that among 10 classes, 33 fields and 62 methods there is no class repeating, 2 fields and 3 methods are repeating. So there are 5 cloned ME out of 105 ME. We get around 5 % clone coverage.

6- RELATED WORK

We observed that research in code clone detection is a well established field [3]. But model clone detection is still at the stage of infancy. We have not mentioned the papers on code clone detection in this section. The reader is advised to go through systematic survey by Rattan et al. [3]. In section 6.1, we mention the studies of model clone detection. The comparison of the proposed work with an existing tool MQ$_{lone}$ is presented in 6.2.

6-1 MODEL BASED CLONE DETECTION

Liu et al. [19] gave an algorithm to detect duplications in UML sequence diagrams by converting 2-dimensional sequence diagram to a one dimensional array. Then a suffix tree is made from the transformed array. The technique detected common prefixes from the suffix tree. On average, 14% duplication is reported in sequence diagrams.

Most of the techniques for model clone detection exist for Matlab/Simulink models. Deissenboeck et al. [1] broke new ground by giving an algorithm to detect clones in graph based Matlab/Simulink models. They convert the model to labeled multi-graph. Maximum weighted bipartite matching algorithm is applied in a breadth first fashion to detect clone pairs. Lastly, repetitive substructures are clustered. The study concludes that one third of model elements are affected by detecting clones in an industrial case study of automotive domain. Deissenboeck et al. [10] explored some challenges being faced in detection of relevant clones in Matlab/Simulink models. The study gives useful insights in addressing those problems too. Pham et al. [20] developed the tool ModelCD to detect clones in Matlab/Simulink models based on two algorithms viz. escan and ascan. escan is used to detect exact matching using an advanced graph matching technique called canonical labeling and ascan is used to detect approximate clones by counting vector of a sequence of nodes and edges' labels.

Hummel et al. [11] presented an incremental algorithm for model clone detection. A directed multigraph is made from the Matlab/Simulink model followed by labeling of relevant edges and blocks. For all subgraphs of the same size, a clone index is calculated. The canonical label of all subgraphs in the clone index is calculated and hashing of similar labels is done.

Deissenboeck et al. [1], [10], Pham et al. [20] and Hummel et al. [11] applied graph isomorphism algorithms for clone detection in Matlab/Simulink models. The underlying approach is transforming the model to a graph structure. This is in contrast to our approach; as graph based techniques do not work for UML models. Transforming a UML class diagram to a graph will make the classes as nodes of the graph. These nodes contain the majority of the information and any graph based approach will not explore these nodes for similarity. Störrle [6] analyzed a number of UML models to support this point of view. Our work converts the classes of UML model into a tree. Then the duplicate subtrees are identified. Our approach differs in principle to Deissenboeck et al. [1], [10], Pham et al. [20] and Hummel et al. [11].

Alalfi et al. [9] customized the code clone detection techniques to detect near miss clones in Matlab/Simulink models. The technique works by transforming the graph-based models to normalized text form and detects clones at different levels of granularity i.e. entire model, sub-system and block level. The authors classified the clones as type-1 (exact) model clones, type-2 (renamed) model clones and type-3 (near-miss) model clones.

There are several studies [21], [22] carried out to detect and visualize differences between versions of UML diagrams. Stephan and Cordy [23] explored various model comparison techniques in a state of the art survey. Rattan et al. [24] presented a technique to detect higher level similarities in source code and between UML models. The technique uses principal component analysis and latent semantic indexing.

An empirical study is carried out by Rattan et al. [25] which showed significant cloning in Matlab/Simulink models. The study used ConQAT [26] to highlight useful patterns of clones in models ranging from 218 blocks to 73888 blocks. Rattan et al. [27] proposed an approach to detect model clones based on tree comparison.

Stephan et al. [28] identified six qualitative areas for evaluation of different Simulink model clone detection approaches viz. relevance and recall, performance, clone detection type, user-interaction required, adaptability, and model pattern granularity. In another study by Stephan et al. [29] on evaluation of model clone detection approaches, artificial clone pairs are created using mutation analysis and injected into the model base of Matlab/Simulink models under study.

6-2 COMPARISON WITH MQ$_{lone}$

Our technique of clone detection starts by exporting a UML class model to an XMI file using the inbuilt facilities of the CASE tool. The core of our technique is the construction of a labeled, ranked tree by carefully mapping the elements parsed from the XMI file to the tree representation such that attribute model elements and method model elements are represented as subtrees. The algorithm to detect duplicate subtrees is applied, which yields the sets of starting positions of repeating subtrees and their lengths. To detect exact and meaningful clones as per the proposed classification, post-processing needs to be applied. The algorithms to calculate frequency of model clones, grouping and clustering are applied to classify the clones. In contrast to our approach, Störrle's MQ$_{lone}$ [6], a first of its kind tool to detect clones in UML domain models is based on model querying. The UML model is exported to an XMI file using any of the UML modeling tools. XMI files are transformed to Prolog files in which clone detection is carried out based on model matching. The result is the prioritized list of matched model elements with their similarity.

In his study, the clones are categorized as Type A (exact model clone), Type B (modified model clone) and Type C (renamed model clone) adapted from code clone classification (Type 1/ Type 2/ Type 3). What's unique about our approach is the classification of model clones, i.e. Type-1 (model clones due to standard modeling/ coding practice), Type-2 (model clones by purpose) and Type-3 (model clones due to design practices.

We understand that our approach will be first of its kind to identify exact and meaningful model clones for development and maintenance purposes. Moreover, we are performing the analysis of forward designed as well as reverse engineered UML class models. Reverse engineered UML class models are open source subject systems which provide the platform for the research community to compare and evaluate the results in the future. These systems have been extensively used in field of code clones to detect, compare and to know the impact of code clones on software quality [3]. On the other hand, MQ$_{lone}$ is evaluated on different UML models created by 16 Master's students. The best model with no natural clones is selected and seeded with artificial clones to emulate Type A, B, and C model clones. Recall and Precision is calculated based on the seeded clones. Different similarity heuristics based on element names and element index are compared on the basis of precision, recall and number of false positives from the subset of detected clones.

MQ_{lone} is applied to different kinds of UML domain model. But our work is focused on detection of exact and meaningful clones in the UML class models at different levels of granularity.

Indeed there is no standard definition of model clone in research community. In code clone domain, the definition of clone is task oriented and dependent on detection technique [30]. Human judgment plays the key role in categorizing what is and is not a clone [31]. The findings of our work in terms of detected clones can be more closely inspected by researchers and practitioners to decide how to deal with them and improve the software design in the future.

7- DISCUSSION

We start this section with a discussion on the findings of the empirical evaluation. The tool reports set of model clones with a frequency more than 2. To find type-1 clones, an inspection is carried out by the authors in this set of model clones. We support our point of view using Stephan's [28] key parameter for model comparison, i.e. "ability to identify recurring patterns using a combination of manual inspection and model visualization".

As an application of the proposed technique in Java programming practice, if a class overrides *equals* method, then it must override the *hashcode* method as well. One can confirm this by inspecting the output of the tool as follows - 1) Set of classes in which equals method repeat. 2) Set of classes in which hashcode method repeats. 3) Compare the two sets for difference.

Thus, one can identify the classes which are not following the above standard programming practice.

We are keen to gain insights into type-3 clones reported by the tool as these will lead to improvement in the design of the system. For instance, in eclipse-ant, we came across interesting clusters repeating in a set of classes. Though these classes inherit the same parent class, but the repetitive clusters are absent in their parent class (As per our interpretation, one reason for the presence of such clusters may be the absence of support of multiple inheritance in Java). We believe that if a UML class model is designed first, then such clones will be detected at an early stage before they appear in an implementation.

Importantly, the tool is able to report model clones of different granularities across the model. In another application, test cases for white box testing are generated from models and most of the logical errors are traced back to design. Therefore, if designs are free from clones then test cases can be designed effectively and efficiently. In a nutshell, the classification and detection of model clones at different levels of granularity is the evidence of the usefulness of the proposed approach.

7-1 THREATS TO VALIDITY

There are several threats to the validity of our study. One potential threat is the definition of term ‘model clone’ and its types. In the field of code clones, the term clone is still searching for the proper definition. A legacy continued in UML models. We have defined the term model clones considering the elements of a class model. But we are of the view that model clones can be defined in other ways also [32].

An external threat to validity is the selection of subject systems for empirical evaluation. Although we chose forward designed and open source reverse engineered systems, we do not know whether these sample systems are representative. There is no standard repository of UML models and real world industrial systems are not available due to different proprietary reasons.

We should also be aware that our tool is based on parsing the XMI file corresponding to the UML class model. Various UML modeling tools export the model to XMI file with varying structure and format. During parsing and construction of the tree, it is difficult to generalize this step for all the modeling tools. It is a definite challenge seeing the number of modeling tools in the market.

We are confident to mitigate the threat to validity regarding classification of clones to some extent. Our classification of model clones relies on the underlying nature of object oriented modeling and do not perceive UML as a notation only.

8- CONCLUSIONS AND FUTURE WORK

Code clones are known to cause several problems in software development and evolution. With the advent of model driven development, clones in UML models will pose similar challenges. Moreover, model clones will propagate to source code, too. Consequently, such a tool to detect clones in UML models is required. In this paper, we have developed a technique to detect clones in UML class models. The technique accepts the XMI file of a UML class model as input. The core of our technique is the construction of a labeled, ranked tree by carefully mapping the elements parsed from the XMI file to the tree representation. The duplicate subtrees are grouped and clustered with the aim to detect exact and meaningful clones. The tool is able to detect clones at different levels of granularity i.e. single attributes/methods of the class, clusters of attributes/methods and complete class in the UML model. A new classification of model clones with the aim to gain some useful insights in the software modeling process is another highlight of the proposed work. The technique has been empirically evaluated on open source reverse engineered and forward designed systems. We believe that the results of the tool are accurate and relevant for practical purposes and demand further investigation.

Currently, the tool is stand alone in nature and detects clones in UML class models only. In the future, work can be carried out to know how type-3 clones can be used to improve the design. UML model refactoring has emerged as an allied area similar to code refactoring. Detection of patterns for model refactoring with the help of our model clone detection approach together with semantic preservation may help in improving the design and structure of the UML class model. We plan to carry such work in future. Further, we are planning to tailor the current technique to apply to other UML models. We hope that the research community would come forward and create an online repository of UML models so that the results of one researcher can be useful to the community. To standardize the term model clone and its types is another thrust area for modeling community. We understand that different UML models have individual features which should be used to develop effective clone detection techniques. Mapping of UML modeling constructs from syntactic to semantic domain will help in detecting semantic model clones.

REFERENCES

[1] F. Deissenboeck, B. Hummel, E. Juergens, B. Schätz, S. Wagner, J-F Girard, S. Teuchert, "Clone Detection in Automotive Model Based Development," International Conference on Software Engineering, pp. 603-612, ACM, New York, 2008.

[2] V. Kulkarni, S. Reddy, A. Rajbojh, "Scaling up Model Driven Engineering – experience and lessons learnt," MODELS Conference, pp. 331-345, ACM New York, 2010.

[3] D. Rattan, R. Bhatia, M. Singh, "Software Clone Detection: A Systematic Review " Information and Software Technology 55 (7), 1165-1199, 2013.

[4] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, "Do code clones matter," International Conference on Software Engineering, pp. 485- 495, ACM, New York, 2009.

[5] L. Jiang, Z. Su, E. Chiu, "Context-based detection of clone-related bugs," ESEC-FSE, pp. 55-64, ACM, New York, 2007.

[6] H. Störrle, "Towards Clone Detection in UML Domain Models," Software and Systems Modeling 12 (2), 307-329, 2013.

[7] OMG: OMG Unified Modeling Language (OMG UML), Superstructure, V2.4.1 (formal/2011-08-05). Tech. rep., Object Management Group, Feb 2011.

[8] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-oriented modeling and design. Addison Wesley, 1991.

[9] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, A. Stevenson, "Model

are Code too: Near Miss Clone Detection for Simulink Models," International Conference on Software Engineering, pp. 295-304, IEEE Computer Society, 2012.

[10] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, B. Schätz, "Model Clone Detection in Practice," International Workshop on Software Clones, pp. 57-64, ACM, New York, 2010.

[11] B. Hummel, E. Juergens, D. Steidl, "Index based Model Clone Detection," International Workshop on Software Clones, pp. 21-27, ACM, New York, 2011.

[12] S. Livieri, Y. Higo, M. Matsushita, K. Inoue, "Analysis of the Linux Kernel Evolution using Code Clone Coverage," Mining Software Repositories, ACM New York, 2007.

[13] A. M. Fernández-Sáez, M. R. V. Chaudron, M. Genero, I. Ramos, "Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: a controlled experiment," Conference on Evaluation and Assessment in Software Engineering, pp. 60-71, ACM New York, 2013.

[14] XMI Guide Version 2.4.1. Tech. rep., Object Management Group. http://www.omg.org/spec/XMI, document number formal/ 2011-08-09, 2011.

[15] N. Göde, B. Hummel, E. Juergens, "What Clone Coverage Can Tell," International Workshop on Software Clones, pp. 90-91, IEEE Computer Society, 2012.

[16] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[17] M. Christou, M. Crochemore, T. Flouri, C. S. Iliopoulos, J. Janoušek, B. Melicha, S. P. Pissis, "Computing all Subtree Repeats in Ordered Trees," Inform. Process. Lett. 112 (24), 958-962, 2012.

[18] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, "Comparison and Evaluation of Clone Detection Tools," IEEE Trans. Softw. Eng. 33(9), 577-591, 2007.

[19] H. Liu, Z. Ma, L. Zhang, W. Shao, "Detecting Duplications in Sequence Diagrams based on Suffix Tree," Asia Pacific Software Engineering Conference, pp. 269-276, IEEE Computer Society, 2006.

[20] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, T. N. Nguyen, "Complete and Accurate Clone Detection in Graph-based Models," Inter-

national Conference on Software Engineering, pp. 276-286, IEEE Computer Society, 2009.

[21] U. Kelter, J. Wehren, J. Niere, "A generic difference algorithm for UML models," Pohl, K., (ed.) Proceedings of National Germ. Conference Software-Engineering 2005 (SE'05), no. P-64. Lecture Notes in Informatics, Gesellschaft für Informatik e.V., pp. 105–116, 2005.

[22] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, P. Zave, "Matching and merging of statecharts specifications," International Conference on Software Engineering, pp. 54–64, IEEE Computer Society, 2007.

[23] M. Stephan, J. R. Cordy, "A Survey of methods and applications of model comparison," Tech. Rep. #2011-582, Queen's University, School of Computing, 43 pp., 2011.

[24] D. Rattan, R. Bhatia, M. Singh, "Detecting High Level Similarities in Source Code and Beyond" International Journal of Energy, Information and Communications 6(2), 1-16, 2015.

[25] D. Rattan, R. Bhatia, M. Singh, "An Empirical Study of Clone Detection in MATLAB/Simulink Models," International Journal of Information and Communication Technology (Accepted).

[26] E. Juergens, F. Deissenboeck, B. Hummel, "CloneDetective – A Workbench for Clone Detection Research," International Conference on Software Engineering, pp. 603- 606, ACM, New York, 2009.

[27] D. Rattan, R. Bhatia, M. Singh, "Model Clone Detection based on tree comparison," Indian Conference, pp. 1041-1046, IEEE CS, 2012.

[28] M. Stephan, M. H. Alalfi, A. Stevenson, J. R. Cordy, "Towards Qualitative Comparison of Simulink Model Clone Detection Approaches," International Workshop on Software Clones, pp. 84-85, IEEE Computer Society, 2012.

[29] M. Stephan, M. H. Alalfi, A. Stevenson, J. R. Cordy, "Using Mutation Analysis for a Model- Clone Detector Comparison Framework," International Conference on Software Engineering, pp. 1261-1264, IEEE Computer Society, 2013.

[30] C. K. Roy, J. R. Cordy, "A Mutation/ Injection-based Automatic Framework for Evaluating Clone Detection Tools" Mutation'09, pp. 157-166, IEEE Computer Society, 2009.

[31] A. Walenstein, N. Jyoti, L. Junwei, Y. Yun, A. Lakhotia, "Problems creating task-relevant clone detection reference data," Working Conference on

Reverse Engineering, pp. 285-29, IEEE Computer Society, 2003.

[32] N. Gold, J. Krinke, M. Harman, D. Binkley, "Issues in Clone Classification for Data flow Languages," International Workshop on Software Clones, pp. 83-84, ACM, New York, 2010.