Chapter 4

Model based Clone Detection

4.1 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 [45]. Model driven software development using UML improves the quality of product delivery and analysis of the product [143]. UML models bring in extra advantages of automatic code generation, early verification and validation [143]. 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. While its counterpart, code clone detection have been an active area of research for long. Evidences suggest that the presence of code clones may lead to bug propagation and maintenance problems [101]. 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 [94]. 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 [216], too.

Key Contributions

Primarily, the objective of the proposed technique 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

Since UML modeling has got inherent object oriented features [232], 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 [198].

Carry out the empirical evaluation on reverse engineered open source systems. Moreover, we are also considering forwarded designed models for evaluation to capture the essence of model driven development.

In this chapter, the background of our work is mentioned in section 4.2. Some elaborative examples to understand our classification of model clones is present in section 4.3 Section 4.4 explain our approach of model clone detection. The empirical evaluation and results of the tool are included in section 4.5. Discussion of the proposed work and threats to validity are mentioned in section 4.6. A brief survey related to work in model clone detection and comparison with existing work is done in section 4.7. Finally, Section 4.8 concludes the work.

4.2 Background

4.2.1 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 [6,45,47,85,187]. Most of these 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 [216] 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 comprising of attributes and operations and relationships among classes. In our approach, we construct a labeled ranked tree from UML class model. The detail of our approach of model clone detection is present in section 4.3.
In a recent study by Saez et al. [57], 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 [9], we evaluated our tool on forward designed and reversed engineered open-source UML class models.

4.2.2 Reasons for model clones

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

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

4.2.3 Definitions

We skip definitions of code clones, types and detection techniques in this chapter. The basic definitions of code clones are given in chapter 2. Definitions necessary for evaluation and understanding of the proposed work are:

**Definition 1 (Model Element)**
A model element refers to a class, an attribute declaration i.e. data type and name of the attribute or a method signature comprising of return type, name of the method and data type of parameters in a UML class model. It is abbreviated as \( ME \) throughout the paper. \#ME refers to total number of model elements extracted from the XMI [239] file of the UML class model.

**Definition 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 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 4 (Clone Cluster)**

A group of model elements present in a model clones 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 4.4, where a detailed evaluation is given for different subject systems.

**Definition 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. 4.1 on library management system, there are 34 model elements. Out of these, 9 are cloned \(ME\). So the coverage % is 26.4%. 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 [159]. By the same token, we believe that clone coverage be applied in the evolutionary study of model clones as well. Göde et al. [68] 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 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.

**4.3 Model clone detection by example**

The objective of our technique is to automatically detect exact and meaningful clones. Thus we propose following categories with explanation using an example.
Consider the sample UML class model of a library management system shown in Fig. 4.1. 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. 4.1 of Library Management System, we notice that field id: Integer is present in most of the classes. It is a modeling practice to uniquely indentify 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/method. Thus the minimum clone length is set to 1 to identify such repeating field/method. Surely, presence of these fields in classes suggests application of programming/modeling practice.

Fig. 4.1 also shows a generalization relationship. The user generalization is shown as Student class and Faculty class to be the subclasses of 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 nutshell, overriding is an important and useful language feature in object oriented programming.

Interfaces are an integral part of UML class diagrams. An interface [199] 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. The figure shows an interface *catalog* with a number of abstract methods. *Catalog* aims to specify the behavior of classes *Manage* and *Search* using realization relationship. Both these classes implement different 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.** In the same diagram, we notice repetition of a cluster of size 3 consisting of attributes *id*: Integer, *name*: String, *address*: String in two classes namely *User* and *Publisher*. This type of repetition in different classes as reported by the proposed technique is categorized as **Type-3-Model clones due to design practices.**

We use the class diagram of Courier Management System shown in Fig. 4.2 as a running example. Among all the classes viz. *CompanyOwner, Customer, Franchisee* and *Employee*, we see repetition of 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. Table 1.1 shows all these clusters together with type of clone. The algorithms to find the clone clusters together with their size are mentioned later in this chapter in section 4.4. The presence of these clusters 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.
Fig. 4.1 Class diagram for Library Management System
Fig. 4.2 A class diagram for courier management system

Table 4.1 Analysis of courier management system

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Name of attribute/method</th>
<th>Freq</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>compensation(): Integer, updateDetail(): void</td>
<td>2</td>
<td>CompanyOwner, Franchisee</td>
</tr>
<tr>
<td>2.</td>
<td>id: Integer</td>
<td>3</td>
<td>Customer, Employee, Franchisee</td>
</tr>
<tr>
<td>3.</td>
<td>Address: String, name: String, phNo: Integer</td>
<td>4</td>
<td>CompanyOwner, Customer, Employee, Franchisee</td>
</tr>
</tbody>
</table>

Coverage % 35% (6/17)

Clone Cluster

<table>
<thead>
<tr>
<th>UID</th>
<th>Size</th>
<th>C (Classes)</th>
<th>Type of clone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>Type-3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>Type-3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>Type-3</td>
</tr>
</tbody>
</table>
Table 4.1 shows the results after applying the proposed technique on the UML class model shown in fig. 4.2. 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 4.1. The table lists the coverage percentage 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.

4.4 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. 4.3 shows the block diagram of the proposed technique.

**Fig. 4.3 Block diagram of the approach**

4.4.1 Modeling and preprocessing

*Modeling:*

We use MagicDraw Enterprise 16.6 CASE tool for creating forward designed UML class models and for reverse engineering open source subject systems. The model is then exported to an XMI file using XMI export facility of the CASE tool. The exported XMI file is input to 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. 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 4.4.1 (Element)**
It is a common term that refers to a package or class or child class or field (attribute) or method (operation).

**Definition 4.4.2 (PackageNode)**
It stores information about a package element, its sub packages and classes.

**Definition 4.4.3 (ClassNode)**
It stores information about a class element, its inner-classes, fields and methods.

**Definition 4.4.4 (ModelTree)**
It is the tree constructed from model elements.

**Definition 4.4.5 (Rank)**
It is the out-degree of a node of the tree.

---

Fig. 4.4 Different nodes of Algorithm 1 as implemented
Fig. 4.4 shows different nodes defined above. These are used in Algorithm 1 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 parameter 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 post-fix order and store it in a list

**4.4.2 Match Detection**

In the pre-processing phase, we constructed the ModelTree from the UML class model.

Fig. 4.5 shows the tree representation of UML class diagram shown in fig. 4.2 of 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. data type node with attribute name as its child node. Various attribute model elements are depicted in red color in fig. 4.5.
Fig. 4.5 Tree representation of the UML class model
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. [37] to compute subtree repetitions. The technique is based on accepting the postfix string representation of 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. 4.5 also shows the postfix string representation of model tree on the right hand side. The rank i.e. out-degree of a node in the tree is shown on the top of the respective node.

**4.4.3 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 which is constructed in pre-processing phase for retrieving the results.

![Fig. 4.6 The input and output of the subtree repeat algorithm [37] for a sample class model](image)
We intend to demonstrate this with an example. A small class diagram having two classes with a pair of repeating attributes is created. Fig. 4.6 shows the actual output of the algorithm [37] 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. $C_i$ 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 $l \in \text{OPL}$
   
   1.1) [RepeatsList] $\leftarrow l$

2. For each $l \in \text{[RepeatsList]}$

   2.1) Count the occurrences, $o$ of $l$ in OPL

   2.2) [NewList] $\leftarrow (l, o)$

   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 $l \in \text{[RepeatsList]}$ && $C_i \in \text{ModelTree}$

   1.1) If $l \in C_i$

      1.1.1) $\text{[GroupingList]} \leftarrow (l, C_1, C_2, .., C_n)$

      1.1.2) $l$ is categorized as field or method

2. For each $C_i \in \text{ModelTree}$ && ($l \in \text{[RepeatsList]}$ && $l \in C_i$)

   2.1) $\text{[RepeatsinClass]} \leftarrow l$

   2.2) $l$ is categorized as field/method

   $\text{[RepeatsinClass]}$ 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 $C_i \in \text{ModelTree}$

   1.1) For each $l \in C_i$ && $l \in \text{[RepeatsinList]}$

   1.2) Store repeating $l$ for corresponding $C_i$

   [Traverse the RepeatsInClass and get a list of repeating members for that Class]

2. For each $C_i \in \text{ModelTree}$

   2.1) Compare $\text{[RepeatsInClass]}$ for $C_i$ in Step-1 to the $C_i$ in this step (Step-2)
3. \([\text{Cluster}] \leftarrow C_i\)

[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.

### 4.4.4 Experimental Setup

The proposed technique has been implemented in Java and runs under Windows 7. It is capable to detect model clones in any object oriented class model. 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 a given algorithm by Christou et al. [37] to compute similar subtrees which has linear time and space complexity. Table 4.2 lists the hardware configuration and software tools needed in implementation.

**Table 4.2 Hardware Configuration and Software Tools**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel (R) Core (TM) i3-3110M CPU @ 2.40GHz 2.40 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installed Memory (RAM)</td>
<td>4.00 GB</td>
</tr>
<tr>
<td>System Type</td>
<td>32-bit Operating System</td>
</tr>
<tr>
<td>Windows Edition</td>
<td>Windows 7 Professional N Service Pack 1</td>
</tr>
<tr>
<td>Integrated Development Environment</td>
<td>• NetBeans IDE</td>
</tr>
<tr>
<td></td>
<td>• Bloodshed Dev-C++</td>
</tr>
<tr>
<td>Software Development Kit</td>
<td>Java Development Kit 6</td>
</tr>
<tr>
<td>Software Modeling Tool</td>
<td>MagicDraw</td>
</tr>
</tbody>
</table>

### 4.4.5 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 4.3 shows the actual measurement of time when the tool is executed on a PC with the configuration as given in table 4.2.

The UML class model is created using 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 Input Transformation**: 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 [37] to computer similar subtrees repeating in the model tree constructed from the class model. The technique is based on accepting the postfix string representation of 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 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 with 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 developer’s viewpoint.
Table 4.3 Execution time

<table>
<thead>
<tr>
<th></th>
<th>eclipse-ant</th>
<th>eclipse-jdtcore</th>
<th>netbeans-javadoc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average of 5 runs</td>
<td>Average of 5 runs</td>
<td>Average of 5 runs</td>
</tr>
<tr>
<td>Preprocessing and Transform</td>
<td>209</td>
<td>322</td>
<td>217</td>
</tr>
<tr>
<td>Computing Subtree Repeats</td>
<td>15</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>Analysis</td>
<td>18</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>Grouping and Reporting</td>
<td>81</td>
<td>30</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>323 ms</td>
<td>393 ms</td>
<td>309 ms</td>
</tr>
</tbody>
</table>

4.5 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 normal development life cycle. Such model helps to check the practical relevance of the proposed approach. Moreover, no standard repository of UML models recognized by modeling community is available; therefore we chose class diagrams created by reverse engineering open source subject systems. It provides a platform for research community to compare and verify the results in future. We chose the subject systems from Bellon’s experiment [21] for empirical evaluation as these systems are well known in code clone detection community.

Subject Systems: The forward designed model is a class model for Proxy Server. This class model is designed using MagicDraw Enterprise 16.6 tool using the standard modeling practices given by OMG [232]. Other subject systems of our study are reverse engineered using the same tool. The characteristics of these systems are listed in Table 4.4. All the systems are in Java with sizes varying from 35K SLOC to 148K SLOC and number of model elements varying from 174 to 292.
Table 4.4 Subject Systems

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Language</th>
<th>Program Size</th>
<th>#ME</th>
<th>XMI File Size (in KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eclipse-ant</td>
<td>Java</td>
<td>35K SLOC</td>
<td>292</td>
<td>947</td>
</tr>
<tr>
<td>eclipse-jdtcore</td>
<td>Java</td>
<td>148K SLOC</td>
<td>174</td>
<td>1445</td>
</tr>
<tr>
<td>netbeans-javadoc</td>
<td>Java</td>
<td>19K SLOC</td>
<td>267</td>
<td>798</td>
</tr>
</tbody>
</table>

Table 4.5 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 4.5 Clones in subject systems

<table>
<thead>
<tr>
<th>Subject System</th>
<th>#ME</th>
<th>Cloned ME</th>
<th>Clones (f=2)</th>
<th>Clones (f&gt;2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eclipse-ant</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classes</td>
<td>30</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>Attributes</td>
<td>106</td>
<td>17</td>
<td>13</td>
<td>04</td>
</tr>
<tr>
<td>Methods</td>
<td>156</td>
<td>28</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>eclipse-jdtcore</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classes</td>
<td>08</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>Attributes</td>
<td>85</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>Methods</td>
<td>81</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>netbeans-javadoc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classes</td>
<td>27</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>Attributes</td>
<td>66</td>
<td>08</td>
<td>05</td>
<td>03</td>
</tr>
<tr>
<td>Methods</td>
<td>174</td>
<td>21</td>
<td>17</td>
<td>04</td>
</tr>
<tr>
<td>Proxy-Server</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Classes</td>
<td>10</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>Attributes</td>
<td>33</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Methods</td>
<td>62</td>
<td>3</td>
<td>00</td>
<td>3</td>
</tr>
</tbody>
</table>
Next, the results of evaluation are presented for all the systems. The results are presented 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 4.2 – 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.

4.5.1 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 4.5 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. 4.7 shows a bar chart showing 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 clone if these are the result of standard modeling/coding practice.

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 relationship between the classes in which these clusters are present.
<table>
<thead>
<tr>
<th>Type of Clone</th>
<th>Type-2 Clone</th>
<th>Type-3 Clone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nature of relationship across classes</td>
<td>Inheritance relationship</td>
<td>Realization relationship</td>
</tr>
<tr>
<td>No. of clones</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**Type-2 clones: Model clones by purpose**

Among type-2 clones, the tool reported 3 clusters repeating across generalization hierarchy in 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 first case, 1 method i.e. `log(String, int): void` and 1 field i.e. `project: Project` is present 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.

**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 %.

98
Fig. 4.7 eclipse-ant

Fig. 4.8 netbeans-javadoc

Fig. 4.9 proxy server
Figs. 4.7 – 4.9 show frequency of cloned ME in different subject systems.

4.5.2 Clones in netbeans-javadoc & 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 4.5. Fig. 4.8 shows 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 repeating 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 4.7, 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 4.7 Type-2 and Type-3 clones in netbeans-javadoc

<table>
<thead>
<tr>
<th>Type of Clone</th>
<th>Type-2 Clone</th>
<th>Type-3 Clone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nature of relationship across classes</td>
<td>Inheritance relationship</td>
<td>Realization relationship</td>
</tr>
<tr>
<td>No. of clones</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

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 bundle: ResourceBundle and 1 method 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.

4.5.3 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. 4.9 shows a bar chart showing repetition of attributes and methods across the model. To analyze the type-1clones, 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.
Clone clusters

In this class diagram, the tool detected 2 clusters. First cluster is made up of three methods namely \texttt{onRequest(HttpServletRequest, HttpServletResponse, URL): void}, \texttt{onRemoteResponse (httpMethod): void} and \texttt{onFinish(): void} repeating in 5 classes \texttt{MimeTypeChecker}, \texttt{HostChecker}, \texttt{MethodsChecker}, \texttt{HostNameChecker}, \texttt{RequestTypeChecker} and 1 interface named \texttt{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 careful application of useful design paradigms. Thus classified as a type-2 clone.

As shown in Table 4.8, there is another cluster made up of 2 methods \texttt{onRemoteResponse (httpMethod): void} and \texttt{onFinish(): void} which is present in the class \texttt{HTTPProxy} in addition to all the five classes mentioned earlier in the first cluster. Since \texttt{HTTPProxy} class has no relationship with any of other class, thus categorized as type-3 clone.

<table>
<thead>
<tr>
<th>Type of Clone</th>
<th>Type-2 Clone</th>
<th>Type-3 Clone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nature of relationship across classes</td>
<td>Inheritance relationship</td>
<td>Realization relationship</td>
</tr>
<tr>
<td>No. of clones</td>
<td>00</td>
<td>01</td>
</tr>
</tbody>
</table>

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 \texttt{ME} out of 105 \texttt{ME}. We get around 5 \% clone coverage.

4.6 Discussion

We start this section with discussion on the findings of the empirical evaluation.

The tool reports set of model clones with 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 et al. [213] key parameter for model comparison i.e. \texttt{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 implementation.

Importantly, the tool is able to report model clones of different granularities across the model. For instance, during empirical evaluation of netbeans-javadoc, we came across instances where classes have identical members i.e. all attributes and operations but different class names. These type-3 clones of class level granularity are relevant to the modeling viewpoint and demand further investigation. Also, Stephan et al. [213, 214] highlighted relevance as one the parameters for comparison and evaluation. 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 nutshell, the classification and detection of model clones at different levels of granularity is the evidence of usefulness of the proposed approach.

4.6.1 Threats to Validity

There are only two potential threats to validity in our study. One potential threat is the definition of term ‘model clone’ and its types. In the field of UML models, the term clone
is still lacking standardization. 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 considering different views of the UML.

An external threat to validity is the selection of subject systems for empirical evaluation. There is no standard repository of UML models and real world industrial systems are not available due to different proprietary reasons.

4.7 Comparative Analysis

An existing tool for clone detection in UML models is $MQ_{clone}$ that presented model clone classification adapted from the work on source code clones [216]. 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 $MQ_{clone}$ [216] tool detects clones in UML domain models and 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). 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 practical purposes. The evaluation in our work provides observations of the
characteristics of model clones which may help in gaining some insights from implementation point of view.

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 to research community to compare and evaluate the results in 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. 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. His models are not publicly available.

Our subject systems are diverse, open source and widely used for development. It may help in understanding the nature of heterogeneous subject systems with respect to cloning. On the other hand, $MQ_{lone}$ is evaluated on a set of homogeneous subject systems. $MQ_{lone}$ 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 [69]. In code clone domain, the definition of clone is task oriented and dependent on detection technique [193]. Human judgment plays the key role in categorizing what is and is not a clone [234]. 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.

4.8 Summary

In this chapter, we have presented a new technique to detect clones in UML class models based on a three stage process. In the first stage, the technique accepts the XMI file of a
UML class model as input. Secondly, a labeled, ranked tree is constructed by carefully mapping the elements parsed from the XMI file to the tree representation. The output of this phase is the list of repeating subtrees. In the third phase of the process, 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. We have proposed a novel classification of model clones keeping in mind the object oriented nature of UML class model such that the detected clones are meaningful for the developer. 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 but the detected clones need to be supported by the intent of software developers regarding relevance for practical purposes. We are currently extending our technique to efficiently refactor type-3 clones.