A Metrics Based Detection of Reusable Object-Oriented Software Components Using Machine Learning Algorithm

Yida Mao
School of Computer Science
McGill University, Montreal

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master

Copyright ©Yida Mao 1999

January 14, 1999
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

Since the emergence of the object technology, organizations have accumulated a tremendous amount of object-oriented (OO) code. Instead of continuing to recreate components similar to existing artifacts, and considering the rising costs of development, many organizations would like to decrease software development costs and cycle time by reusing existing OO components. The difficulty of finding reusable components is that reuse is a complex and thus less quantifiable measure. In this research, we first proposed three reuse hypotheses about the impact of three internal characteristics (inheritance, coupling, and complexity) of OO software artifacts on reusability. Corresponding metrics suites were then selected and extracted. We used C4.5, a machine learning algorithm, to build predictive models from the learning data set that we have obtained from a medium sized software system developed in C++. Each predictive models was then verified according to its completeness, correctness and global accuracy. The verification results proved that the proposed hypotheses were correct. The uniqueness of this research work is that we have combined the state of the art of three different subjects (reuse detection and prediction, OO metrics and their extraction, and applied machine learning algorithm) to form a process of finding interesting properties of OO software components that affect reusability.
Résumé

Depuis l'apparition de la technologie des objets, les organisations ont accumulé une grande quantité de code orienté objets (OO). Au lieu de continuer à recréer des composants semblables aux composants existants, et vu les coûts croissants de développement, beaucoup d'organisations voudraient diminuer des coûts et la durée du cycle de développement du logiciel en réutilisant les composants existants. La difficulté de détection des composants réutilisables réside dans le fait que la réutilisation reste une activité complexe et par conséquent difficilement mesurable. Dans ce travail de recherche, nous proposons d'abord trois hypothèses sur l'impact de trois caractéristiques internes (héritage, couplage et complexité) sur la réutilisabilité des composants OO. Pour chaque hypothèse, nous avons sélectionné et extrait un ensemble de métriques. Nous avons ensuite utilisé C4.5, un algorithme d'apprentissage symbolique, pour construire des modèles prédictifs à partir des données d'apprentissage extraites d'un système de taille moyenne écrit en C++. Nous avons validé chaque modèle en calculant sa complétude, son exactitude et sa pertinence globale. Les résultats ont prouvé que nos hypothèses sont correctes à des degrés différents. L'originalité de ce travail de recherche réside dans le fait que nous avons combiné les travaux sur trois domaines différents (la réutilisation, les métriques OO et l'apprentissage symbolique automatique) pour définir un processus de détection des propriétés des composants OO qui affectent la réutilisabilité.
Acknowledgment

I praise the only wise God who created me and gives me wisdom and perseverance to finish this research work. For many times, He showed His mercy to me when I was depressed, tired, and stressed. May all the glories and honors be to our almighty Lord.

I would like to thank my parents for their loving kindness and sacrifice. They created a golden opportunity for me so that I could pursue higher education in Canada. Their encouragement and guidance lead me through many difficulties and hardness during all these years that I am abroad. They have demonstrated the greatest love I have ever experienced in my whole life. I also feel grateful for my dear wife, Cory, who cared me through the time of this research. Her positive and cheerful personality brought me a lot of happiness.

I also want to thank Dr. Houari Sahroui and Dr. Hakim Lounis, who are my supervisors from Centre de recherche informatique de Montreal (CRIM). I have learned quite a lot from their knowledge and experience in Software Engineering research. I really enjoyed working with both of him because they were always there when I needed help. They were also very humble and approachable. Three of us worked as a team all the time. I am glad that our paper based on this research was accepted by the thirteenth IEEE Automated Software Engineering Conference held in October, 1998 in Hawaii, U.S.A.. Professor Gerald Ratzer helped me on the part of thesis preparation and writing. I thank him for correcting many errors occurred in the first draft of this thesis because of my limited skill in written English. I really admire him because he always makes himself available to the students went they need him. Such a professor is not seen very often these days.

I appreciate two secretaries: Ms. Franca Cianci and Ms. Judith Bracke, who supported and helped me during the research work and the thesis preparation.

Finally, I would like to dedicate this thesis to my beloved parents and my wife.
Contents

1 Introduction .............................................. 1
   1.1 Detection of reusable components ....................... 2
       1.1.1 Detection with specification ....................... 2
       1.1.2 Detection without specification ................... 3
   1.2 Motivation and aim .................................... 4
       1.2.1 Benefit of reuse .................................. 4
       1.2.2 Aim ............................................. 5
   1.3 Thesis organization .................................... 6

2 Background and Related Work .............................. 7
   2.1 An overview of software metrics ....................... 7
       2.1.1 Class internal metrics .......................... 8
       2.1.2 Class external metrics ......................... 9
       2.1.3 System level metrics ............................ 10
       2.1.4 Class relationship metrics ..................... 10
       2.1.5 Inheritance coupling metrics .................... 14
   2.2 Detection of software properties ..................... 15
   2.3 Machine learning languages ........................... 17
3 Specific Problem Statement

3.1 Difficulties of reuse detection ........................................... 18
3.2 Misconceptions of reuse .................................................. 20
3.3 The problem .................................................................. 21

4 Solution Strategy .................................................................. 23

5 Experimental Frame Work .................................................... 25

5.1 Making reuse hypothesis .................................................. 25
  5.1.1 Inheritance vs. Reuse .................................................. 26
  5.1.2 Coupling vs. Reuse .................................................... 27
  5.1.3 Complexity vs. Reuse ................................................... 28

5.2 Metrics selection ............................................................. 29

5.3 Metrics extraction ............................................................ 32

6 Building models of evaluation and detection .......................... 34

6.1 Building decision trees in C4.5 .......................................... 34
6.2 Simplify decision trees in C4.5 .......................................... 35
6.3 Converting trees to rules ................................................... 39

6.4 Options to enhance the predictive model ............................. 42
  6.4.1 Windowing option ....................................................... 42
  6.4.2 Grouping option ........................................................ 43
  6.4.3 Confidence factor ....................................................... 44
  6.4.4 Weight option .......................................................... 44

6.5 Building the best predictive models .................................... 44
6.6 Data preparation ................................................. 46
  6.6.1 Dependent variable vs. Independent variables ................. 47
  6.6.2 The C4.5 input files ........................................ 48

6.7 Selection of the model building options ............................. 50

7 Experimental Results and Validations ............................... 53
  7.1 The predictive models ......................................... 53
    7.1.1 Hypothesis 1 .............................................. 53
    7.1.2 Hypothesis 2 (code metrics) ............................... 55
    7.1.3 Hypothesis 2 (design metrics) ............................. 56
    7.1.4 Hypothesis 3 .............................................. 57

7.2 Validation of the predictive models ................................ 58
    7.2.1 Validation method .......................................... 58
    7.2.2 Validation tool ............................................ 59
    7.2.3 Validation results .......................................... 59

7.3 Discussion ..................................................... 64

8 Conclusion and Future Work ........................................ 66
  8.1 Contributions .................................................. 66
  8.2 Future work ..................................................... 67

A A validation and consulting tool: *Reusability Consult* .................... 71
List of Tables

2.1 Coupling between Hierarchies ........................................ 13
2.2 Summary of metrics suites ........................................... 15
6.1 Options run with C4.5 for selecting the best predictive model ... 45
6.2 A composition of criteria for selecting the best predictive models ... 46
6.3 Dependent and independent variables .................................. 48
6.4 Complexities and error rates of models for hypothesis 1 ........ 51
6.5 Complexities and error rates of models for hypothesis 2 (design metrics) ... 51
6.6 Complexities and error rates of models for hypothesis 2 (code metrics) ... 52
6.7 Complexities and error rates of models for hypothesis 3 ........ 52
7.1 Four-class classification performance matrix ..................... 59
7.2 Validate hypothesis 1 using training data ....................... 60
7.3 Validate hypothesis 2 (design metrics) using training data ........ 61
7.4 Validate hypothesis 2 (code metrics) using training data ........ 61
7.5 Validate hypothesis 3 using training data ....................... 61
7.6 Validate hypothesis 1 using testing data .......................... 62
7.7 Validate hypothesis 2 (design metrics) using testing data ........ 62
7.8 Validate hypothesis 3 using testing data ................. 62
7.9 Validate hypothesis 1 using testing data, altered presentation ....... 63
7.10 Validate hypothesis 2 using testing data, altered presentation ....... 64
7.11 Validate hypothesis 3 using testing data, altered presentation ....... 64
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The waterfall model</td>
<td>24</td>
</tr>
<tr>
<td>6.1</td>
<td>A decision tree before pruning</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>A pruned decision tree with its estimated error rate</td>
<td>39</td>
</tr>
<tr>
<td>6.3</td>
<td>Portion of decision tree presented in Figure 6.1</td>
<td>40</td>
</tr>
<tr>
<td>6.4</td>
<td>Rule set of decision tree presented in Figure 6.1</td>
<td>41</td>
</tr>
<tr>
<td>6.5</td>
<td>C4.5 name file that defines reuse classification and attributes for hypothesis 1</td>
<td>49</td>
</tr>
<tr>
<td>6.6</td>
<td>A portion of C4.5 data file for hypothesis 1</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>Predictive model for hypothesis 1 in rule based model</td>
<td>54</td>
</tr>
<tr>
<td>7.2</td>
<td>Predictive model for hypothesis 2 (code metrics) in rule based model</td>
<td>55</td>
</tr>
<tr>
<td>7.3</td>
<td>Predictive model for hypothesis 2 (design metrics) in rule based model</td>
<td>56</td>
</tr>
<tr>
<td>7.4</td>
<td>Predictive model for hypothesis 3 in rule based model</td>
<td>57</td>
</tr>
<tr>
<td>A.1</td>
<td>The structure and functionality of RC</td>
<td>71</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Since the emergence of the objects technology, organizations have accumulated tremendous amount of object-oriented (OO) code[16]. Instead of continuing to recreate components similar to existing artifacts, and considering the rising costs of development, many organizations would like to decrease software development costs and cycle time by reusing existing OO components to reduce the time and the human effort required to build software products.

Software reuse is the process of using existing software artifacts instead of building them from scratch[4]. Generally speaking, there are three steps involved in such kind of reuse process:

1. Selecting a reusable component from available resource.
2. Adapting it to the purpose of the new system under development.
3. Incorporate it into the software system being developed.

Although each step plays an equally important role during the process of reuse, a poorly chosen off-the-shelf component can often lead the subsequent steps into inevitable disasters that may eventually ruin the goal of reuse. Thus, one of the important issues in the reuse business is what to reuse. How do we identify a piece of object-oriented software code which will finally lead to a successful reuse? This research attempts to solve this problem.
1.1 Detection of reusable components

There are many reasons for searching for reusable components. Two of them are the most common. The first reason is to find a reusable artifact that is similar to what is currently being built. Thus, people know what kind of specific components they are looking for. Software developers who build ATM switches are more interested in a semaphore class that supports multi-thread agents than a math library which provides complicated matrix operations. We refer this kind of reusable components searching as detection with specification. In the second case, people are solely interested in looking for those artifacts that are potentially reusable. Criteria of finding such components are not data abstractions or specifications of new components under construction, but rather the reusability lying in the candidate components. We call this way of finding reusable artifacts as detection without specification. The similarities and differences between these two differently motivated queries of reusable components are better explained in the following two sections.

1.1.1 Detection with specification

In software engineering projects, reuse occurs frequently as an informal sharing of techniques and products among people developing the same or similar projects[10]. When looking for reusable components, software developers have specifications of the components to be built, possibly with object/class design documentation and/or data abstractions. These requirements serve as query criteria against which all candidate components are evaluated. Software developers select candidate components either from systems/projects previously developed by themselves or from software components repository/library containing matrix operations, statistical packages, operating system primitives, and basic data structures and their operations[13]. Quality components that have similar functionality, close data abstractions and parallel structures are most likely to be chosen.

Here is an example of software reuse in the development environment of OO. We have a class named \textit{DateInterval} in an application of management of employee's holidays. Some of its primitives are:

- Given the beginning \textit{date} and the end \textit{date} of a period represented by an interval, verifying if a \textit{date} is included in this interval.  
- Verifying if two intervals overlap, if so, return the intersection of the two intervals as another interval.
A new class called *TimeInterval* of *time* is needed for an application of task planning in a factory. Two of the most important primitives that we are going to build are:

- Given the beginning *time* and the end *time* of a period represented by an interval, verifying if a *time* is included in this interval.

- Verifying if two *time* intervals overlap, if so, return the intersection of the two intervals as another *time* interval.

*TimeInterval* basically provide the same primitives as those of *DateInterval* except its intervals consist of *time* rather than *dates*. Software developers can either choose to implement *TimeInterval* from scratch or to make a reuse of the existing *DateInterval* class. Taking a close look at these two classes, one can find that the main difference between the two classes is the type of the attributes representing the beginning and the end of the intervals. We can reuse *DateInterval* by changing the attributes of beginning and end of interval from *date* to *time*. Further adaptation and verification effort need to be spent in order to incorporate *DateInterval* into the new system. To some software developers, reuse is a trivial approach in this example. If time and effort could be saved, people would rather reusing existing components[29]. In the real world, the process of reuse is far more complicated than the scenario presented here. But the basic steps of reuse are almost the same.

### 1.1.2 Detection without specification

Currently, most reuse occurs in the project development cycle[10], where reuse is rather passive because it is driven by project-specific activities whose aims are system delivery. Packaging reusable components will never be the primary concern. In order to accomplish higher levels of reuse, we should separate the process model that support project-specific activities from reuse-packaging activities. The latter is referred as a *component factory*[10]. It detects potentially reusable components and packages them so they are easy for the project-specific activities to use. The component factory provides code components to the project development model upon demand, and generates and maintains a repository of software artifacts for future use. Therefore, the component factory’s process model performs two tasks: it answers requests for reusable components coming from the project development model, meanwhile it also prepares itself for satisfying those requests. In order to answer efficiently requests from the project development model, the component factory has to prepare itself by produce some software components without
specific requests from the project development model. It extracts reusable components from existing systems or generalizes components previously produced on request from the project development model.

We use the example presented in section 1.1.1 to illustrate this concept. In this particular example, the component factory first recognizes the `DateInterval` class as a reusable class and then packages it by parameterizing the interval attribute (e.g., using templates in C++), so that the interval class can be reused by different (unknown) applications in the future. If a project requests a component that satisfies a given specification, e.g., as of the ones we listed for `TimeInterval`, `DateInterval` will be returned by the component factory for concern.

The detection of reusable components occurs when the component factory propagates. The identification is solely based on the reusability of the candidate components without being influenced by any class/object specifications or data abstractions. Therefore the reusability of a candidate component, e.g., `DateInterval`, is not evaluated by how functionally or structurally it is close to another component in demand, e.g. `TimeInterval`, but rather by the possibility and potentiality of its being reused by different (unknown) applications in the future.

1.2 Motivation and aim

Detection of potentially reusable components is a vital starting point in building a reusable component factory. Reusing an appropriate ready-made software artifact facilitates the development process meanwhile enhances overall product quality. However, adopting a mistakenly chosen component not only increases development effort but also invokes extra maintenance cost. A major hindrance that prevents software developers from adapting the reuse technology is there exists no well defined and empirically validated set of methodology that guides them to find potentially reusable code components with confidence. This research is motivated by the benefits of reuse technology. Its aim is to find a more systematic and efficient methodology of detecting reusable object-oriented software component.

1.2.1 Benefit of reuse

Software reuse is considered to be one of the most promising approaches for increasing productivity[15]. By reusing existing software, programmer can save the effort of
re-implementation. Besides potentially increasing productivity and reducing the development cycle time, the reuse of proven components (particularly verbatim reuse artifacts that have been thoroughly tested) should reduce the maintenance burden and lead to more reliable and efficient systems. Tracz[27] used an analogy that compares used cars to used programs. Unlike high-mileage used cars, software never fatigues. The more users, the more systems it is reused, and the longer it has been reused, the more desirable a software artifact is. An empirical study conducted by Basili et al.[4] found significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity in the context of OO systems.

OO languages provide a more flexible environment for component reuse than conventional programming languages. Compared to functionally oriented programming, OO approaches make reuse more efficient for the following reasons[16]:

- An existing class can have new objects as instances of another class. This is similar to but more general than library reuse for conventional program.
- A subclass of an existing class can be defined as supplying additional or specialized functionality by classes that reuse the inherited functionality of the ancestor class.
- Parametric polymorphism allows objects to respond to similar commands in different ways by using types as method parameters. The most important advantage of parametric polymorphism is code sharing, which is a form of reuse. The reuse example given in section 1.1.1 is a good illustration of this scenario.

Demanded by modern distributed and cooperative execution environments, OO languages provide a high degree of flexibility in both usage and adaptation. Using properly, OO can limit adaptation efforts because it has a specialization mechanism and allows developers to concentrate on those objects and methods that need modification.

1.2.2 Aim

We are inspired by the empirically validated evidence of how properly conducted reuse can influence productivity and quality in OO systems[4, 15]. We are particularly interested in the early phase of building a component factory, which is to detect reusable components without specifications. In this phase, program units are extracted, made independent, and measured according to observable properties related to their potential for reuse. The aim of this research is to search for the tangible and quantifiable properties that affect the
reusability of OO components, to emphasize the relation between the isolated properties and reusability, and finally to produce a predictive model that assists software developers to identify reusable components. This predictive model must be empirically validated.

1.3 Thesis organization

The rest of the thesis is organized as follows. The next chapter introduces the background and the related work of software reuse, software metrics, and some applications of the machine learning languages. In Chapter 3, we present the research problem in a more specific way. Chapter 4 proposes the solution strategy that are employed in this research. The experimental framework is then described in Chapter 5. Chapter 6 explained the C4.5 machine learning language that is used to build the predictive models which help us to verify the reuse hypotheses. The predictive models and their verification results are presented and discussed in Chapter 7. In the last chapter, we make some conclusions to this research work and list some of the work that we would like to carry in the future. The Appendix A briefly explains the structure and the functionality of the validation and consulting tool that is build for this study.
Chapter 2

Background and Related Work

This research consists of three different domains: software metrics, software reuse, and machine learning languages. In this chapter we introduce the background of these three fields that are related to reuse detection. A major portion of this chapter is dedicated to software metrics and their capabilities of detecting and measuring various interdependencies and attributes of software artifacts. Related works of detecting reusable components are introduced thereafter. At the end, we present some of the research work that use machine learning models to estimate various software properties.

2.1 An overview of software metrics

Software metrics assess specific attributes of software components, and generally deal with specific static characteristics of the components. By using metrics, we can assess specific internal and external component attributes and study the dependencies between components and their environment. We first introduce an intuitive method that classifies various metrics before presenting all the software metrics that have been surveyed in this work.

Henderson-Sellers[18] classified the metrics according to various perspectives of an OO system. These perspectives are:

- **Class internal measure**: measures class size and complexity. These measures happen inside a class.
- **Class external measure**: measures the interface a of class externally at class level, e.g., the number of services it provides.

- **System level measure (ignoring relationships)**: accumulates measures from the two previous categories. For instance, the total number of classes in a system.

- **Class relationships (excluding inheritance)**: measures coupling between objects and classes. Couplings due to inheritance are not counted here.

- **Inheritance coupling measure**: particularly measures the inheritance hierarchy of a system and its complexity.

Metrics in the first three categories can be used to study the attributes and characteristics of code components other than interdependencies which are specifically measured by metrics in the latter two categories. This metrics classification method helps us to better understand the purpose and usage of different metrics which are presented in the following five sections.

### 2.1.1 Class internal metrics

**Weighted Methods Per Class (WMC)**: this is one of the six metrics proposed by Chidamber and Kemerer[12]. Consider a class \( C_1 \) with methods \( M_1, \ldots, M_n \) that are defined in the class. Let \( c_1, \ldots, c_n \) be the complexity of the methods, then

\[
WMC = \sum_{i=1}^{n} C_i
\]

If all method complexities are equal to unity, the \( WMC = n \), or the number of methods. The authors add that the complexity metric to be used here was deliberately not specified to allow for the most general application of the metric.

**Response For Class (RFC)**: is the size of the Response Set of a class, defined as the set of methods in the class together with the set of methods called by the class's methods[12].

**Number Of Methods (NOM)**: the number of local methods in a class[21].

**Number Of Properties (NOP)**: the number of attributes plus the number of local methods[21].
Class Interface size (CIS): counts the number of public methods in a class.

Number of Polymorphic Methods (NOP): counts of the methods that can exhibit polymorphic behavior, e.g., virtual methods in C++.

Number of Parameters Per Method (NPM): represents the average of the number of parameters per method in a class. It is computed by summing the parameters of all methods and dividing by the number of methods in a class.

Number of Attributes (NOA): counts the number of attributes in a class.

Number of Abstract Data Types (NAD): counts the number of user defined objects, i.e. abstract data types, used as attributes in a class and are therefore necessary to instantiate an object instance of the class.

Number of Reference Attributes (NRA): counts the number of pointers and references used as attributes in a class.

Number of Public Attributes (NPA): counts the number of attributes that are declared as public in a class.

Class Size in Bytes (CSB): is the size of the objects in bytes that will be created from a class declaration. The size is computed by summing the size of all attributes declared in a class.

2.1.2 Class external metrics

Depth of Inheritance Tree (DIT): counts the depth of a class in the inheritance tree; if multiple inheritance is involved, the the depth of the class is the length of the maximum path from the node representing the class to the root of the tree. The depth of the root class is 0.

Number of Children (NOC): counts the number of immediate subclasses subordinated to a class in the class hierarchy.

Number of Ancestors (NOA): counts the number of distinct classes which a class inherits.
2.1.3 System level metrics

System Size in Classes (DSC): counts the total number of classes in the system.

Number of Hierarchies (NOH): counts the number of class hierarchies in the system.

Number of Independent Classes (NIC): counts the number of (standalone) classes that are not inherited by any classes in the system.

Number of Single Inheritance (NSI): counts the number of classes (sub classes) that use single inheritance in the system.

Number of Multiple Inheritance (NMI): counts the number of classes that use multiple inheritance in the system.

Number of Leaf Classes (NLC): counts the number of leaf classes in the hierarchies of the system.

Average Depth of Inheritance (ADI): is the average depth of inheritance of classes in the system. It is computed by dividing the summation of maximum path lengths to all classes by the number of classes. The path length to a class is the number of edges from the root to the class in an inheritance tree representation.

Average Width of Inheritance (AWI): is the average number of children per class in the system. The metric is computed by dividing the summation of the number of children over all classes by the number of classes in the system.

Average Number of Ancestors (ANA): is the average number of classes from which a class inherits information. This metric is similar to the ADI measure and differs only when there are instances of multiple inheritance in the system.

2.1.4 Class relationship metrics

The Chidamber and Kemerer (C & K) metrics suite is the most cited and also the most criticized set of metrics. Basili et al.[4] show that five of the six C & K metric were useful in predicting class fault-proneness during the high and low level design phases of the life cycle. The metrics were found to be statically independent and did not capture a great deal of redundant information. They conclude that the C & K metrics proved to be better predictors than the best set of the traditional metrics, which are only available at the latter phases of the software life cycle. The following two metrics were proposed by Chidamber and Kemerer[12].

Coupling Between Object class (CBO). CBO for a class is a count of the number of other classes to which it is coupled, where coupling is defined as any evidence of a method of one object using methods or instance variables of another object.

Lack of COhesion in Methods (LCOM). LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero, where similarity of a pair of methods is the number of joint instance variables used by both methods.

For instance, consider a class C with three methods $M_1$, $M_2$, and $M_3$. Let $I_1 = \{a, b, c, d, e\}$, $I_2 = \{a, b, e\}$, and $I_3 = \{x, y, z\}$, where $I_i$ is the set of instance variables used by method $M_i$. There are two disjoint sets: $I_1 \cap I_2 = \{a, b, e\}$ and $I_3$. There is one pair of methods who share at least one instance variable ($I_1$ and $I_2$). Therefore, $LCOM = 2 - 1 = 1$.

Lounis et al.[22] proposes a comprehensive suite of measures to quantify the level of coupling in modular software systems. Different kinds of coupling can appear in modular software systems written in C/C++ programming language. The authors have precisely defined a set of coupling measures so that several forms of coupling can be determined algorithmically. The authors consider a module as a collection of units, collected in a file and its associated header. A program unit is one or more contiguous program statements having a name by which other parts of the system can invoke it. There are two different kinds of modules interconnections:

- **Common interconnection** If the modules are to be used together in a useful way, there may be some external references, in which the code of one module refers to a location in another module. This reference may be to a data location defined in one module and used in another.

- **Unit-call interconnection** An entry point of a unit appears in the code of one module and is called from another module unit.

The distinction between different kind of modules interconnection is distinguished by these three criteria:

- The kind of information shared by interconnected modules: parameters or global areas.
- The type of shared information: scalar, structure, class,...
What use is made with this shared information. Uses can be classified into the following three categories:

- **C-use** happens when a variable is used on the right side of an assignment statement or in an output statement.
- **P-use** occurs when a variable is used in a predicate statement.
- **I-use** occurs when a variable is used in an assignment to another variable, and this latter variable is then used in a predicate statement.

Here are some of the metrics derived from the combinations of the aforementioned criteria:

- **No Parameters Interconnection (NPI):** m calls n or n calls m. No passing parameters, common variables references, or common references to external media.

- **Scalar-Control Interconnection (SCI):** Some scalar variable in m is passed as an actual parameter to n where it has a P-use.

- **Scalar-Reference Data-Control Interconnection (SRDCI):** The address of a scalar variable in m is passed as an actual parameter to n where it has an I-use, but no P-use.

- **Global-Data Interconnection (GDI):** m and n share references to the same global variable. This latter is defined and used in n and C-used in m. It would be possible that this variable is not visible to the entire system.

All 24 metrics are empirically validated and a rule of predicting of fault-proneness is derived from C4.5.

Price et al.[23] believe it is more appropriate to evaluate the criteria at the class hierarchy level of abstraction and to study the reusability of a class hierarchy as a who, portions of a class hierarchy, or a set of related class hierarchies. Thus, a class in a system can be categorized either as a **General** class or a **Specific** class. A **General** class is one that is expected to be reused in other applications. A **Specific** class is a class that is only applicable in this application.

While General and Specific classes provide a characterization mode within an inheritance hierarchy, it is also important to identify the interactions between the hierarchies.
that comprise an OO application. These interactions provide the important first step in discerning the couplings between classes when viewed from the perspective of entire hierarchies. Thus, to augment the General/Specific classes, the software designer is asked to define the class hierarchies that are related to one another in an OO application. A hierarchy is defined as related to another hierarchy if they are related in concept and are expected to be reused together in future systems. Relating class hierarchies encourages the designers to group their components into reusable portions at the earliest stages in the design process. Based on the above two categorizing methods, there are eight different kind of relationship between two classes in a system:

The authors goal is to direct the software designer to strive for maximum reuse by organizing all coupling into $G \rightarrow G$, if they are in related hierarchies, or $S \rightarrow S$, if they are in unrelated hierarchies.

<table>
<thead>
<tr>
<th>Coupling type</th>
<th>Hindrance to reuse?</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \rightarrow G$ among related hierarchies</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>$G \rightarrow G$ among unrelated hierarchies</td>
<td>Yes</td>
<td>Attempt to move the dependency to their Specific descendant classes that are most relevant. Create new classes if necessary.</td>
</tr>
<tr>
<td>$G \rightarrow S$ among related hierarchies</td>
<td>Yes</td>
<td>Attempt to move the destination to an appropriate General ancestor class.</td>
</tr>
<tr>
<td>$G \rightarrow S$ among unrelated hierarchies</td>
<td>Yes</td>
<td>Attempt to move the source to an appropriate Specific descendant class.</td>
</tr>
<tr>
<td>$S \rightarrow G$ among related hierarchies</td>
<td>No</td>
<td>Attempt to move the source to an appropriate General ancestor.</td>
</tr>
<tr>
<td>$S \rightarrow G$ among unrelated hierarchies</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>$S \rightarrow S$ among related hierarchies</td>
<td>No</td>
<td>Attempt to move both the source and destination to appropriate General ancestor classes.</td>
</tr>
<tr>
<td>$S \rightarrow S$ among unrelated hierarchies</td>
<td>No</td>
<td>None.</td>
</tr>
</tbody>
</table>

Table 2.1: Coupling between Hierarchies
2.1.5 Inheritance coupling metrics

Briand et al. [9] proposed a comprehensive suite of measures to quantify the level of class coupling during the design of OO systems. This suite takes into account the different OO design mechanisms provided by the C++ language (e.g., friendship between classes, specialization, and aggregation). There are three different facets, or modalities, of coupling between classes in OO systems developed with C++. They are referred as locus, type, and relationship:

- **Relationship** refers to the type of relationship: friendship, inheritance, or other (neither). Clearly, a class \( C \) is most closely coupled with all its descendants, ancestors, friends. The following functions help to define these relationships:
  
  - \( \text{Friends}^{-1}(C) \) is a function that returns the set of classes that have class \( C \) as a friend.
  
  - \( \text{Ancestors}(C) \) is a function that returns the set of classes that are the ancestors of \( C \). Ancestors refers to the base classes of \( C \), and their base classes, and so on (closure).
  
  - \( \text{Friends}(C) \) is a function that returns the set of classes that are the friends of \( C \).
  
  - \( \text{Descendants}(C) = \text{Systems}(S) - \text{Friends}(C) - \text{Descendants}(C) - \text{Friends}^{-1} - \text{Ancestors}(C) - \{C\} \).

  - **Locus** refers to expected locus of impact; i.e., whether the impact of change flows towards a Class (import) or away from a Class (export). A Class \( C \) exports impact to its friends and descendants, and imports impact from its ancestors and classes that have \( C \) as their friend.

  - **Type** refers to the type of interactions between classes (or their elements): It may be Class-Attribute (CA) interaction, Class-Method (CM) interaction, or Method-Method (MM) interaction.

Coupling between classes in C++ can be due to any combination of these facets. Using measures that can account for all different types of interactions, we can evaluate the actual impact of each coupling dimension on the quality of the resulting artifact. As can be seen above, we have three types of relationship, two loci, and three types of interactions.
Considering all combinations, we have 18 different possible types of coupling measures such as friendship attribute interaction export, ancestor method interaction import, and so on. Here is the summary of all the resulting metrics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Interactions</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFCAIC: Inverse Friend CA Import Coupling</td>
<td>ACA($C_i, C_j$)</td>
<td>Friends$^{-1}$</td>
</tr>
<tr>
<td>ACAIC: Ancestors CA Import Coupling</td>
<td>ACA($C_i, C_j$)</td>
<td>Ancestors</td>
</tr>
<tr>
<td>OCAIC: Others CA Import Coupling</td>
<td>ACA($C_i, C_j$)</td>
<td>Others</td>
</tr>
<tr>
<td>FCAEC: Friends CA Export Coupling</td>
<td>ACA($C_i, C_j$)</td>
<td>Friends</td>
</tr>
<tr>
<td>DCAEC: Descendant CA Export Coupling</td>
<td>ACA($C_i, C_j$)</td>
<td>Descendants</td>
</tr>
<tr>
<td>OCAEC: Others CA Export Coupling</td>
<td>ACA($C_i, C_j$)</td>
<td>Others</td>
</tr>
<tr>
<td>IFCMIC: Inverse Friend CM Import Coupling</td>
<td>ACM($C_i, C_j$)</td>
<td>Friends$^{-1}$</td>
</tr>
<tr>
<td>ACMIC: Ancestors CM Import Coupling</td>
<td>ACM($C_i, C_j$)</td>
<td>Ancestors</td>
</tr>
<tr>
<td>OCMIC: Others CM Import Coupling</td>
<td>ACM($C_i, C_j$)</td>
<td>Others</td>
</tr>
<tr>
<td>FCMEC: Friends CM Export Coupling</td>
<td>ACM($C_i, C_j$)</td>
<td>Friends</td>
</tr>
<tr>
<td>DCMEC: Descendant CM Export Coupling</td>
<td>ACM($C_i, C_j$)</td>
<td>Descendants</td>
</tr>
<tr>
<td>OCMEC: Others CM Export Coupling</td>
<td>ACM($C_i, C_j$)</td>
<td>Others</td>
</tr>
<tr>
<td>IFMMIC: Inverse Friend MM Import Coupling</td>
<td>AMM($C_i, C_j$)</td>
<td>Friends$^{-1}$</td>
</tr>
<tr>
<td>AMMIC: Ancestors MM Import Coupling</td>
<td>AMM($C_i, C_j$)</td>
<td>Ancestors</td>
</tr>
<tr>
<td>OMMIC: Others MM Import Coupling</td>
<td>AMM($C_i, C_j$)</td>
<td>Others</td>
</tr>
<tr>
<td>FMMEC: Friends MM Export Coupling</td>
<td>AMM($C_i, C_j$)</td>
<td>Friends</td>
</tr>
<tr>
<td>DMMEC: Descendant MM Export Coupling</td>
<td>AMM($C_i, C_j$)</td>
<td>Descendants</td>
</tr>
<tr>
<td>OMMEC: Others MM Export Coupling</td>
<td>AMM($C_i, C_j$)</td>
<td>Others</td>
</tr>
</tbody>
</table>

Table 2.2: Summary of metrics suites

Based on actual project defect data, the hypotheses underlying these coupling measures are empirically validated by analyzing their relationship with the probability of fault detection across class. The results demonstrate that some of these coupling measures may be useful early quality indicators of the design of OO systems.

2.2 Detection of software properties

Software metrics help to better study the relationships between measurable properties of OO components and reusability. Caldiera et al. [10] used a suite of four metrics which
evaluate software artifacts against their usefulness, costs, and quality. The case studies indicated that highly reused components had volume and complexity lower than those less reused components. Price et al.[23] presented a technique to analyze and measure the reusability of OO designs. All classes in OO systems are classified as either General (application independent, which is more reusable) or Specific (application dependent, which is less reusable). Assuming that a General class is potentially more reusable than a Specific class, the General classes must be towards the top and the Specific classes must be towards the bottom of the class hierarchy. It is because the descendants of a Specific class are also have to be Specific. Relationships between class hierarchies are also defined to be either Related (expected to be reused together) or Unrelated (not expected to be reused together). A set of eight metrics were defined on the combination of the above two classification methods. These metrics helps to evaluate OO systems from reuse point of view. For instance, a dependency from a General class to another General class in a related hierarchy is good for reuse, while a dependency from a General class to a Specific class in a related hierarchies is bad for reuse.

Etzkorn et al.[16] approached the problem of detecting reusable code in a very unique way. They believed that comments and identifiers reflect human concepts more clearly than algorithms which are implemented in the source code. An automated system was built to examine code comments to extract information on reusability. They considered aspects of the language in which the comments were written, particularly syntax and grammar. Since half of the comments were in sentence form, the system could apply natural language parsing, which could combine a syntactical approach using a natural language parser and a semantic approach using a knowledge base. With help of software quality metrics, this system could provide a thorough analysis of how potentially reusable a component was.

Cost of rework of reusable software is another key criterion of selecting potentially reusable components. Basili et al.[5] conducted a study to model and understand the cost of rework in a library of reusable software components. A predictive model, which was based on a set of metrics, of impact of error source on rework effort was built. Li and Henry[21] focused their study on the relationship between metrics prediction models and maintenance effort. They found that maintenance effort could be predicted from combinations of metrics collected from source code of OO components.
2.3 Machine learning languages

Building and validation of predictive models of varicus OO properties are enriched by aid of Machine-Learning (ML) techniques. The aforementioned predictive model of [5] was validated and enhanced by C4.5 in [26]. Instead of building predictive models to estimate reusability, they tried to quantify software correction costs. The overall results demonstrated that the predictability of the machine learning algorithms were better than models built by neural networks and least-square regression.

A more detailed study in exploring the potential of ML based approach to estimate software rework was conducted by Almeida et al.[1]. They have empirically investigated different machine learning techniques with regard to their capabilities to generate accurate correctability models. Four very well known, public-domain machine learning algorithms, which are NewID, CN2, C4.5 and FOIL, have been studied. These algorithms were compared with regard to their capabilities to assess the difficulty of correct Ada faulty components from some systems. Predictive models were produced by these machine learning algorithms, respectively. The predictability of each model was then evaluated by sensitivity, specificity, accuracy and etc..
Chapter 3

Specific Problem Statement

Detection of reusable software component is not a new topic for computer science researchers. Software reuse has been known for improving development productivity and product quality[5]. Effective reuse of ready-made software artifacts can decrease development cost and thus reducing project delivery time. However, not every organizations succeeded in applying reuse technology. Four case studies conducted by Fichman et al. indicated that adopting reuse in OO environment was unpredictable and hardly have a clear and easily achieved result[17]. Before presenting the specific problem that we try to solve, let us look at some of the difficulties that people sometimes encounter when trying to conduct software reuse. Nevertheless some of them are due to misunderstanding.

3.1 Difficulties of reuse detection

Research and experiments of software reuse has been conducted for almost two decades. Back in 1968, M.D. McIlroy envisioned the future of used-program market and thus proposed using modular software units to encourage reuse in software developments[27, 10]. In early 1980's, Wasserman et al.[28] predicted that reuse technologies would be widely used by software engineers in the 1990's when software development would rely heavily on prefabricated software components. By then software engineers would be more concerned about overall system structure and connections between existing code[29]. It is the late 1990's, the popularity of software reuse is unfortunately still not as high as predicted.

Although OO programs are potentially more reusable than functionally oriented code, most OO systems were not designed nor developed with the concept of reuse[16]. A
candidate component in a well designed and developed OO system may not be reusable at all because it is too rigid and fragile to be adapted into a new environment[13]. The difficulties of retrieve potentially reusable components are large. To determining the reusability of a candidate component, the following questions are often asked:

- **Is the code component structurally close to the new component under development?** The structural (or syntactic) distance measures to what extent the two components look alike[20]. This question can be interpreted in plain English as: are the purpose and capabilities of the code component useful in the current domain[16]? Unfortunately, converting a structurally reusable component to a functionally reusable component is not a simple task, especially when the candidate component deeply depends on other non-reusable components.

- **Is the quality of the code component sufficient enough?** If a software artifact conforms to certain standards, its reusability is increased because of the perceived quality and usability of the software. Software developers should be confident in what they are about to reuse. This kind of confidence is not gained through objective estimations but by measures from empirically verified quality indicators. Nevertheless, we have to realize that a piece of useful program with good coding quality are not necessary reusable. Today, a lot of programmers are able to produce bulletproof code, but unable to use it afterwards.

- **How difficult it is to isolate the artifact from its environment?** Some components are difficult to reuse because they are highly dependent on other components that are not needed. Tremendous amount of effort will be spent on separate the desired components from those portions on which they depend. Software developers often give up using such components because the cost of separation exceeds the original cost of building new components from scratch. The ease of isolating a piece of useful code might be one of the primary criterion that people should consider when come to reuse. We should be aware that the cost of extracting a code component is not a simple measure. It is impossible to evaluate the effort of separating a deeply interconnected component from its original system.

- **How much effort is saved by reuse?** Some measures were proposed to evaluate the cost or effort saved due to reuse. One of them is the gain ratio which measures the reuse benefit of a project or system, as being the normalized (percentage) financial gain due to reuse[15]. A simple measure of reuse benefit $R_b$ of a system $S$, in terms of development cost ($C$) can be calculated as:

$$
R_b(S) = \frac{C(S_{\text{without reuse}}) - C(S_{\text{with reuse}})}{C(S_{\text{without reuse}})}
$$
This formula solely concerns the cost of development that excludes the incremental benefit to revenue from the product. It will be a lot easier if we can acquire such a gain ratio when identifying reusable components. But an insightful thought tells us that to predict the gain ratio is not easier than to calculate the reusability itself.

The questions listed above can be viewed as the basic criteria for detecting reusable code components. There are definitely more requirements and standards besides these concerns. Except for the first one, the other three questions are all applicable to detection of reusable components with and without specifications. Nevertheless, the answers to some of the questions are difficult to acquire. Unless we have the answers to all of these questions, an individual answer to any of them is inadequate for determine reusability.

3.2 Misconceptions of reuse

Despite the difficulties of reuse, sometimes people misunderstand the purpose of reuse and how to reuse. Jacobson et al.[19] listed some typical misconceptions regarding software reuse that mislead management, software designers and engineers from proper software reuse. Here are some typical excuses or concerns that we may hear very often[19]:

- **Reuse does not work.** Disasters of reusing existing software in several organizations should not blind the fact that there are more organizations successfully made reuse work. People who doubt about the effectiveness of reuse often would like to ask this question: how much can be saved by using existing software components when developing new software systems[15]? With more and more organizations increasing in reusing software, it is necessary to demonstrate to the management that reuse make good business sense by showing clear financial evidence of the benefits of reuse in real development processes. However, it is hard to track directly the actual cost saved due to reuse. We can always ask each developers in an project that supports reuse technology about how much time or effort that they have saved by reusing existing code in order to estimate the financial benefit of reuse. But such kind of information is likely to be unreliable and inconsistent[15]. In our opinion, reusability or cost of rework is so intangible that they can be hardly measured or predicted. However, by studying the properties of OO components whose reusability are already known, we are able to build a predictive model that detects reusability based on something that is tangible and quantifiable.

20
Adopting reuse technology is too expensive. Adopting reuse technology is not cheap, especially for creating systematic reuse. Reuse is not just a matter of programmers. A single programmer can only reuse part of his or her own code in a very small percentage. This kind of individual reuse does not impact on the overall system development processes at all. It is the matter of reengineering an organization's business processes. Setting up a reuse business costs money. But it should more than return the initial investment and also coin money down the trail. The component factory model described in section 1.1.2 helps to reduce the cost of reuse. Instead of searching through existing programs looking for reusable component, the component factory decreases the amount of expensive human analysis needed in the software development process by limiting analysis to components that are really worth considering[10]. Therefore, the phase of detecting reusable components should base on some well-defined and empirically validated predictive model which can be automated eventually.

Developers will select components to be reused from existing software artifacts when necessary. Woodfield et al. conducted an experiment to test if programmers could accurately assess component reusability[29]. As the matter of fact, software developers without proper training and practice cannot assess the worth of reusing a candidate artifact to satisfy the implementation requirement of a specified software component under development. Their decisions were influenced by some unimportant features like size of the candidate components in SLOC (Source Lines Of Code). Furthermore, if people can not save more than 30 percent of the effort of creating a new component by reuse, they had rather start to write them from scratch. For that reason, a systematic detection model is in demand to help to build a component factory from where the software developers can request reusable components. Nevertheless, such a detection model should provide consistent and trustworthy result so that developers working in different domains do not get contradictory predictions.

3.3 The problem

What kind of observable properties will help us to predict the reusability of a software component? It is obvious that a software component is reusable if the effort required to reuse it is remarkably smaller than the effort required to implement a component with the same functions[24]. We believe reusability is a complex characteristic that pertains to certain properties of a code component. Some components are more reusable than others. Thus we would like to emphasize the relationship between properties of code components
and their reusability. The problem we try to solve in this research is broken down to the following more detailed questions:

1. What kind of dependencies and properties of OO components that affect reusability? In the context of OO programming, a component is rarely independent from its environment. We are particularly interested in finding out the relationship between reusability and interdependence of classes and objects. We also believe that some components are easier to reuse than others due to some internal attributes that can be measured purely in terms of the components themselves and not with respect to how they relate to their environment[8], e.g., code complexity.

2. How to detect and measure those dependencies and properties that affect reuse? Software metrics have been widely used to measure the software product and the process by which it is developed. Metrics can be applied to any artifact constructed during development, including not only code, but also analysis and design models as well as their components[13]. Those metrics that detect and measure the dependencies and properties found for the previous problem will be identified.

3. How precise and accurate are the measures of identified dependencies and properties as indicators of reusability? We need to verify them empirically. We would like to investigate the usefulness and significance of the measures obtained in (2.) by using some validation methodologies proposed in[8]. We are also interested in building a predictive model that helps software developers to analyze the candidate code components and to predict how reusable they are.

Since reusability is a complex measure we need to evaluate it by assessing some simpler and quantifiable properties of OO components. Using these tangible measures, we are able to study how to predict reusability based on something that is more trivial. In the next chapter, we propose the detailed approach to solve the above problems.
Chapter 4

Solution Strategy

We decide to take the following steps in order to solve the problems stated in the previous chapter:

1. *Reusability hypotheses proposal*: we propose hypotheses regarding the relationship between reusability and measurable properties of OO components. Some of the quantifiable properties of OO artifacts are levels and degrees of inheritance, various types of coupling measures, and numerous kinds of code complexity measures.

2. *Metrics selection*: according to the reusability hypotheses proposed above we select the appropriate metrics to detect and measure the properties that are related to reusability. Good candidates for these metrics are those metrics suites that are empirically verified and commonly cited.

3. *Building predictive models*: the relationship between reusability and the selected properties of OO components can be validated by building some predictive models using the selected machine learning languages, e.g., C4.5[26] to see how tightly the properties of OO components are related to reusability. We will select the best predictive models based on its error rates. Different options of C4.5 will be applied with the model construction process in order to choose the best model.

4. *Verification and evaluation*: the correctness, completeness and accuracy of the predictive models will be verified and evaluated. It is achieved by applying the predictive models to OO systems to predict the reusability of their components. The predicted reusability will then be compared to the actual reusability assessed by system administrators with strong background of software reuse. At this stage we are able to judge the correctness of the hypotheses proposed at the very beginning.
These four steps form an interesting waterfall model which looks similar to the more common software development cycle model. Each step in this model depends on its previous step. And the result of the last step, which is validation and evaluation, can be feedback to the beginning part of the model to refine the whole process. In the following chapters, we are going to describe the work that we have accomplished for each of these four steps in a very detailed way.
Chapter 5

Experimental Frame Work

This chapter describes the work accomplished to achieve the first two solutions proposed in the previous chapter. We first introduce the process of making reuse hypotheses. According to these hypotheses, several metrics suites are then chosen to detect the properties of OO components, which are considered to be useful in predicting reusability. Metrics values are extracted with aid of some OO code analyzing tool. The relationship of reusability and measurable properties of OO components is then emphasized by building predictive models using the selected machine learning algorithms, i.e., C4.5, which is introduced in Chapter 6.

5.1 Making reuse hypothesis

Reusability is a complex measure, which is domain dependent. Some components are more reusable in one domain than in others[23]. Our goal is not to search for a set of methods measuring reusability universally, but to study some specific aspects and characteristics pertaining to OO programming languages, e.g., C++, that affect reusability. In the following three sections, we propose three reusability hypotheses regarding the relationships between reuse and inheritances, dependencies, and complexity, respectively.

In this research, we restrict the unit of a reusable component to be a Class containing attributes and methods. Detection of reusable abstract data structures, individual methods, or reusable sub-systems (groups of classes) is beyond the scope of this study.
5.1.1 Inheritance vs. Reuse

A class hierarchy is the result of the inheritance relationship between classes of a system. Inheritance, which is emphasized in OO programming languages, is a binary, asymmetric connection between two classes [13]. Class inheritance enables objects and methods of one class to be accessed by another class, which inherits the former. The class from which the inheritance is taken is called a superclass or parent class, and the class that inherits other class(es) is referred as a subclass or child class. Classes containing the common objects and methods are inherited by subclasses where specific objects and methods are defined. As a result of inheritance, classes with more general objects and methods are usually located in the upper part of the hierarchy meanwhile classes dealing with specific operations are consequently lowered down to the hierarchy.

Hypothesis 1

Price et al.[23] claimed that classes in the top portion of a hierarchy, which are more desirable in many other systems, are potentially more reusable than the lower level classes, which are more domain specific. All classes in a system can be identified as either General or Specific according to its purpose and flexibility. A General class is a class that is able to provide services beyond the context that it is defined within. A Specific class is a class that is only applicable in the current system. Therefore, the General classes must be towards the top and the Specific classes must be towards the bottom of the class hierarchy in order to encourage reuse and to facilitate the process of reuse [23]. It seems logical that the deeper a Specific class is in the class hierarchy, the more General class properties this class can access because of its inheritance, making it more complex to predict its behavior [12]. Li and Henry also empirically verified that understanding the classes in the lower hierarchy are more difficult. Therefore the classes in the lower hierarchy are more likely to be misused because they depend critically on too many other classes [21]. Obviously, effort of separating a class that depends heavily on other classes in the system are high.

Difficulties of understanding a component that highly depends on other components compounds the cost of separating it from its environment. The higher the unpredictability of the cost to isolate a component the less likely it is chosen for reuse. In addition, deep class hierarchies usually imply problems of conceptual integrity and thus complicates the process of selecting reusable components [4].

Based on the aforementioned experimental work, we propose the following hypothesis:

26
**Hypothesis 1**: A component's position in the class hierarchy of the system within which it is defined somehow affects its reusability.

5.1.2 Coupling vs. Reuse

The degrees of interdependence among the components of a software system is usually referred as Coupling, which is defined as any evidence of a method of one object using methods or instance variables of another object[12]. It is believed that low coupling measures indicate good quality in software design and implementation [9, 22, 12].

**Hypothesis 2**

A class is rarely independent from the system within which it is defined. It offers services and needs services from other components. Intuitively, a class is more reusable when its dependence on other components of the system is weak. In other words, the less a class needs services of other components, the more reusable. Some of the hindrances to reuse brought by interdependencies between classes are explained in the following paragraphs.

Strong interrelated components are harder to understand. Comprehending such components inevitably invokes understanding other parts of the system. Sometimes, understanding a heavily interdependent component can initiate a cascade of comprehending related components. The cost of rework and benefit of reuse can be hardly estimated when the effort of understanding starts to chain-react. Software engineers sometimes are unwilling to reuse classes that tightly depend on other classes because of such unpredictability.

If selected for reuse, such components are also difficult to extract when they highly depend on other components that are not desired [13]. In such a case, a potentially useful class requires tremendous effort to be separated from its environment that is not needed. When the cost of extraction becomes too high, reuse is most likely aborted.

Not surprisingly, components with high coupling measures tend to be more fault-prone than components that are less dependent [4, 9, 22]. The former usually requires more effort of testing and verification before they are fully integrated into a new environment. Furthermore, it is also believed that the high level of interconnection between classes is a significantly impediment to maintainence[7, 6]. Summing up all the uncertainties, unpredictability, and risks encountered when reusing highly dependent classes, reuse hardly gains any advantages over implementation from scratch.
Based on the above discussion, we propose the following hypothesis:

**Hypothesis 2:** The dependencies lie between a component and its environment within which it is defined somehow affect its reusability.

### 5.1.3 Complexity vs. Reuse

Measures of complexity and volume have been validated as being capable of predicting the maintenance effort and the cost of rework in reusing software components [21, 5]. An empirical study conducted by Caldiera et al. further indicated that small and simple code components often minimize the cost of searching and extraction [10]. The cost of reusing a class, which includes extraction and adaptation, is highly influenced by its understandability and readability, which can be indicated by measuring its volume and complexity.

**Hypothesis 3**

Complexity influences reuse in three different aspects. First, a component has to be understood before it is extracted, modified, and finally reused. The complexity and the understandability of designs are closely related. Typically, the more complex the component, the harder it is to understand it [21]. The high measures obtained in volume and complexity of a class implies its low readability, which affects its understandability. Therefore, measures of volume and complexity provide a partial indication of how easily a component is qualified for reuse [10].

Secondly, a class with a large number of methods is likely to be more application specific, which reduces its reusability as discussed in the previous hypothesis [12]. A class with large amount of contents does not necessarily imply its usefulness. Contrarily, classes with a large number of methods can sometimes represent poorly designed classes amalgamating a lot of unrelated methods [3]. As a result, high complexity makes the quality and fitness of the candidate class less certain.

Thirdly, a complicated code component also demands a greater level of understanding required on the part of the testing and debugging of the class [12]. Thus, the cost of using such components is high because high complexity and low readability makes modification and integration more difficult. As a result, the adaptation cost becomes highly unpredictable.

**Hypothesis 3:** A component's volume and complexity somehow affects its reusability.
5.2 Metrics selection

Appropriate metrics should be selected from those empirically verified and commonly cited metrics suites for detecting and measuring aspects of code components that affect reusability. We decide to use different metrics suites to verify our hypotheses:

Hypothesis 1: A component's class hierarchy somehow affects its reusability. The following metrics are used to measure components position in class hierarchy and various attributes related to inheritance.

- **Depth of Inheritance Tree (DIT):** measures the position of a class in the inheritance hierarchy[21]. The depth of a class is the maximum length from the node to the root of the tree[12]. The root class's DIT is zero.

- **Height of Inheritance Tree (HIT):** measures the position of a class in the inheritance hierarchy from another point of view. The height of a class is the maximum length from the node to a leaf node of the tree. A leaf node's HIT is zero.

- **Number Of Ancestors (NOA):** counts the number of distinct classes which a class inherits.

- **Number Of Children (NOC):** is the number of immediate subclasses subordinated to a class in the class hierarchy[12]. It counts the number of subclasses that inherit the methods of the current class.

- **Number of inherited methods (oim):** measures the number of methods that are used in the current class, but are defined in its ancestor classes.

- **Number of inherited variables (oiv):** is the number of attributes that are used in the current class but are defined in its ancestor classes.

Hypothesis 2: The dependencies lie between a component and its environment within which it is defined somehow affect its reusability. Coupling is an interesting and complex relationship between OO classes and their encompassing objects and methods. There are many ways to obtain numerous coupling measures[9, 13, 12, 3, 22]. However we decide to verify hypothesis 2 using two different coupling measures: design coupling and code coupling.

The whole suite of metrics proposed by Briand et al. measuring various of couplings between classes from OO design point of view is selected[9]. Detailed definition of these metrics is presented in section 2.1.5.
- Inverse Friend CA Import Coupling (IFCAIC)
- Ancestors CA Import Coupling (ACAIC)
- Others CA Import Coupling (OCAIC)
- Friends CA Export Coupling (FCAEC)
- Descendant CA Export Coupling (DCAEC)
- Others CA Export Coupling (OCAEC)
- Inverse Friends CM Import Coupling (IFCMIC)
- Ancestors CM Import Coupling (ACMIC)
- Others CM Import Coupling (OCMIC)
- Friends CM Export Coupling (FCMEC)
- Descendant CM Export Coupling (DCMEC)
- Others CM Export Coupling (OCMEC)
- Inverse Friends MM Import Coupling (IFMMIC)
- Ancestors MM Import Coupling (AMMIC)
- Others MM Import Coupling (OMMIC)
- Friends MM Export Coupling (FMMEC)
- Descendant MM Export Coupling (DMMEC)
- Others MM Export Coupling (OMMEC)

We also select all 24 coupling metrics by Lounis et al. which are used to measure coupling occur at code level[22]. Detailed definition of these metrics is described in section 2.1.4.

- No Parameters Interconnection (NPI)*
- Scalar-Data Interconnection (SDI)
- Return-Data Interconnection (RDI)*
- Stamp-Data Interconnection (StDI)
- Scalar-Control Interconnection (SCI)
- Return-Control Interconnection (RCI)*

1Metrics marked with asterisk were verified to have impact on predicting fault-proneness of measured components in [22].
- Stamp-Control Interconnection (StCI)
- Scalar-Data/Control Interconnection (SDCI)
- Return-Data/Control Interconnection (RDCI)*
- Stamp-Data/Control Interconnection (StDCI)
- Tramp Interconnection (TI)
- Scalar-Reference Data Interconnection (SRDI)*
- Scalar-Reference Control Interconnection (SRCI)*
- Scalar-Reference Data-Control Interconnection (SRDCI)
- Scalar-Reference Modification Interconnection (SRMI)
- Stamp-Reference Data Interconnection (StRDI)*
- Stamp-Reference Control Interconnection (StRCI)
- Stamp-Reference Data-Control Interconnection (StRDCI)
- Stamp-Reference Modification Interconnection (StRMI)
- Global-Data Interconnection (GDI)
- Global-Control Interconnection (GCI)
- Global-Data/Control Interconnection (GDCI)
- Global-Modification Interconnection (GMI)
- Type Interconnection (TyI)*

**Hypothesis 3:** A component’s volume and complexity somehow affects its reusability. The following metrics are selected to measure the volume and complexity of candidate classes.

- **Weighted Methods Per Class (WMC):** this is one of the six metrics proposed by Chidamber and Kemerer[12]. Consider a class C1 with methods $M_1, \ldots, M_n$ that are defined in the class. Let $c_1, \ldots, c_n$ be the complexity of the methods, then

$$WMC = \sum_{i=1}^{n} C_i$$

If all method complexities are equal to unity, the WMC = n, or the number of methods. The authors add that the complexity metric to be used here was deliberately not specified to allow for the most general application of the metric.
- **Response For Class (RFC):** is the size of the Response Set of a class, defined as the set of methods in the class together with the set of methods called by the class's methods[12].

- **Number Of Methods (NOM):** the number of local methods in a class[21].

- **Class Interface size (CIS):** counts the number of public methods in a class.

- **Number of Parameters Per Method (NPM):** represents the average of the number of parameters per method in a class. It is computed by summing the parameters of all methods and dividing by the number of methods in a class.

- **Number Of Properties (NOP):** the number of attributes plus the number of local methods[21].

- **Number of Polymorphic Methods (NOP):** counts of the methods that can exhibit polymorphic behavior, e.g., virtual methods in C++.

- **Number of Attributes (NOA):** counts the number of attributes in a class.

- **Number of Abstract Data Types (NAD):** counts the number of user defined objects, i.e. abstract data types, used as attributes in a class and are therefore necessary to instantiate an object instance of the class.

- **Number of Reference Attributes (NRA):** counts the number of pointers and references used as attributes in a class.

- **Number of Public Attributes (NPA):** counts the number of attributes that are declared as public in a class.

- **Class Size in Bytes (CSB):** is the size of the objects in bytes that will be created from a class declaration. The size is computed by summing the size of all attributes declared in a class.

### 5.3 Metrics extraction

Metrics extraction can be a tedious task if not aided by language analyzing tools. A typical language tool processes source code, and produces some sort of output. C++ programming tools are available which extract information about the organization of a program for example, class hierarchies, cross reference lists, class complexities, or call graphs. The two extraction tools that are selected and used in this research are GEN++[14] and QMOOD[2].
GEN++ is an application generator for creating code analyzers for C++ programs. Developed by Prem Devanbu[14, 9] and his team, GEN++ are used to extract the whole suite of design coupling metrics selected for validating hypothesis 2. QMOOD (which stands for Quality Model for Object-Oriented Designs represented in C++) is designed and developed to support object-oriented design quality assessment. It has been developed so as to make it easy for the hierarchical quality model to be applied to real-work designs. QMOOD is used extensively in this research for extracting both metrics suites selected for hypothesis 1 and hypothesis 3. However we are unfortunately not able to find a more efficient way to extract the code coupling metrics for hypothesis 2. Thus, the detection and measure of the whole suite of code coupling metrics are performed manually, which is very time consuming.
Chapter 6

Building models of evaluation and detection

In order to verify the reusability hypotheses proposed in chapter 5, we need to build some characterization models. The purposes of constructing such models are twofold. First, we use these models to emphasize the relationship between reusability and specific properties of OO components. The predictability of such models is to be studied and thus strengthened afterwards. Second, these models can be used to easily assess class reusability based on their type and their level of inheritance, coupling, or complexity. They can also help software engineers better understand what coding standards should be avoided or enforced in order to increase reusability. The model building technique that we used is a machine learning algorithm called C4.5[26]. In the following sections, we describe how we built the characterization models.

6.1 Building decision trees in C4.5

Most of the work done in machine learning has focused on supervised machine learning algorithms. Starting from the description of classified examples, these algorithms produce definitions for each class. In general, they use an attribute-value representation language that allows the use of statistical properties on the learning set. They are helpful for constructing efficient software quality models. C4.5 belongs to the divide and conquer algorithms family. In this family, the induced knowledge is generally represented by a decision tree. It is the case of algorithms like ID3 (the C4.5 ancestor[25]), CART[11]. The principle of this approach could be summarized by this algorithm:
If the examples are all of the same class
Then - create a leaf labeled by the class name;
Else - select a test based on one attribute;
     - divide the training set into subsets, each associated to one of the possible values of the tested attribute;
     - apply the same procedure to each subset;
EndIf

The key step of the algorithm above is the selection of the "best" attribute to obtain compact trees with high predictive accuracy. Information theory-based heuristics have provided effective guidance for the division process.

C4.5 induces Classification Models, also called Decision Trees, from data. ID3, C4.5's ancestor, works with a set of examples where each example has the same structure, consisting of a number of attribute/value pairs. One of these attributes represents the class of the example (could mention dependent variable/independent variable here). Usually the class attribute take only the values true, false, or success, failure or something equivalent.

A decision tree can be used to classify a case by starting at the root of the tree and moving through it until a leaf is encountered. At each nonleaf decision node, the case's outcome for the test at the node is determined and attention shifts to the root of the subtree corresponding to this outcome. When this process finally leads to a leaf, the class of the case is predicted to be that recorded at the leaf.

A decision tree is important not because it summarizes what we know, i.e. the training set, but because we hope it will classify new cases correctly. Thus, when building classification models, one should have both training data to build the model, and test data to verify how well it actually works. We present both the characterization models built from the training data set and the verification of these models using the test data set in the next chapter.

6.2 Simplify decision trees in C4.5

The recursive partitioning method of constructing decision trees described in section 6.1 will keep subdividing the set of training cases until each subset in the partition contains cases of a single class, or until no test offers any improvement. The result is often a very complex decision tree, which is not only hard to comprehend, but also proven to be more
error prone in prediction[26]. A decision tree is not usually simplified by deleting the whole tree in favor of a leaf. Instead, the idea is to remove parts of the tree that do not contribute to classification accuracy on unseen cases, producing something less complex and thus more comprehensible.

Basically, there are two ways to modify the recursive partitioning method in order to grow simpler trees:

- deciding not to divide a set of training cases any further, or
- removing retrospectively some of the structure built up by recursive partitioning.

The first method, known as **stopping** or **prepruning**, saves time in assembling structure that is not used in the final simplified tree. The typical approach is to look at the best way of splitting a subset and to assess the split from the point of view of statistical significance, information gain, error reduction, or etc.. If this assessment falls below some threshold, the division is rejected and the tree for the subset is just the most appropriate leaf. However, this approach may not work as expected: too high a threshold can terminate division before the benefits of subsequent splits become evident, while too low a value results in little simplification. Therefore C4.5 favors the second approach in which an overfitted tree is pruned after it is produced.

Decision trees are usually simplified by discarding one or more subtrees and replacing them with leaves; as when building trees, the class associated with a leaf is found by examining the training cases covered by the leaf and choosing the most frequent class. In addition, C4.5 allows replacement of a subtree by one of its branches. Let us assume that we are able to predict the error rate of a tree and of its subtrees. We can trim a decision tree using the following method. Start from the bottom of the tree and examine each nonleaf subtree. If replacement of this subtree with a leaf, or with its most frequently used branch, would lead to a lower predicted error rate, then prune the tree accordingly. This affects the predicted error rate of all trees that encompass the trimmed subtree. This process will lead to a tree whose predicted error rate is minimal with respect to the allowable forms of pruning because the error rate for the whole tree decreases as the error rate of any of its subtrees is reduced. However we still have to resolve the question of how to predict the error rate of a tree.

There are two kinds of techniques used to predict error rate. The first one is to predict the error rate of a tree using a new set of cases that is distinct from the training set. Since these cases were not seen at the time the tree was built, the estimates obtained from them are obviously unprejudiced and trustworthy if there the size of the test set is large enough.
Unfortunately, this is not always the case. The drawback of this approach is that a portion of the available data must be reserved for the separated set, so the original tree must be constructed from a smaller training set. This may not be much of a disadvantage when data is abundant, but can lead to a more serious problem when the data set is relatively small.

The approach taken in C4.5 uses only the training set from which the tree was built. It is clear that error rate on the training set from which the tree was built does not provide a suitable estimate. One way around this problem is to use a cross-validation approach. In this approach the available data set is divided into $N$ equal-sized blocks so as to make each block's number of cases and class distribution as uniform as possible. For each block, a tree is constructed from cases in all the other blocks and tested on cases in the reserved block. For moderate values of $N$, the assumption is made that the tree constructed from all but one block will not differ much from the tree constructed from all data. $N$ different classification models are then built. Provided that $N$ is not too small\(^1\), the average error rate over the $N$ unseen test sets is a good predictor of the error rate of a model built from all the data.

Let's take a look at an example. Figure 6.1 shows the original output of a decision tree derived from a training set of 300 cases\(^2\). This dataset records the votes of all United States congressmen on 16 key issues selected by the *Congressional Quarterly Almanac* for the second session of 1984. The unpruned decision tree consists of 25 branches. The $(N)$ or $(N/E)$ appearing after a leaf indicated that the leaf covers $N$ training cases, $E$ erroneously. For instance, the following decision path:

```
-physician fee freeze=y:
  -synfuels corporation cutback=y:
    -duty free exports=n:
      -education spending=y:republican
```

covers 13 training cases, unfortunately, 2 of them are classified as *republican* instead of *democrat* by this decision tree. The simplified decision presented in Figure 6.2 contains only 7 branches and 5 leaves. Notice the $(N/E)$ presented after a leaf. The $N$ still indicates the number of cases covered by following a path from the root to a certain leaf. However the $E$ is not the exact number of cases that are misclassified. It is a compound *estimated pessimistic error rate*\(^3\) calculated during the process of pruning.

---

\(^{1}\)10 is a common number.

\(^{2}\)This example was included in the sample data directory of the original C4.5 software.

\(^{3}\)Refer to section 4.2 in [26] for detailed calculation.
Decision Tree:

physician fee freeze = n:
  | adoption of the budget resolution = y: democrat (151.0)
  | adoption of the budget resolution = u: democrat (1.0)
  | adoption of the budget resolution = n:
  |   | education spending = n: democrat (6.0)
  |   | education spending = y: democrat (9.0)
  |   | education spending = u: republican (1.0)
physician fee freeze = y:
  | synfuels corporation cutback = n: republican (97.0/3.0)
  | synfuels corporation cutback = u: republican (4.0)
  | synfuels corporation cutback = y:
  |   | duty free exports = y: democrat (2.0)
  |   | duty free exports = u: republican (1.0)
  |   | duty free exports = n:
  |     | education spending = n: democrat (5.0/2.0)
  |     | education spending = y: republican (13.0/2.0)
  |     | education spending = u: democrat (1.0)
physician fee freeze = u:
  | water project cost sharing = n: democrat (0.0)
  | water project cost sharing = y: democrat (4.0)
  | water project cost sharing = u:
  |   | mx missile = n: republican (0.0)
  |   | mx missile = y: democrat (3.0/1.0)
  |   | mx missile = u: republican (2.0)

Figure 6.1: A decision tree before pruning

which is substituting subtrees with branches or leaves. Therefore, most of the values of E are real float numbers.

The summary of results on the training cases tested using both the original tree and the pruned tree is presented in Figure 6.2. The Errors are the results of the number of wrongly classified cases divided by the size of the training set. The sum of the predicted errors at the leaves, divided by the number of cases in the training set, provides an immediate Estimate of the error rate of the pruned tree on unseen cases. For this tree, the sum of the predicted errors at the leaves is 20.8 for a training set of size 300. By this estimate, then, the pruned tree will misclassify 6.9% of unseen cases. Although the

38
Simplified Decision Tree:

physician fee freeze = n: democrat (168.0/2.6)
physician fee freeze = y: republican (123.0/13.9)
physician fee freeze = u:
| mx missile = n: democrat (3.0/1.1)
| mx missile = y: democrat (4.0/2.2)
| mx missile = u: republican (2.0/1.0)

Tree saved

Evaluation on training data (300 items):

<table>
<thead>
<tr>
<th>Before Pruning</th>
<th>After Pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size Errors</td>
<td>Size Errors</td>
</tr>
<tr>
<td>25 8 (2.7%)</td>
<td>7 13 (4.3%)</td>
</tr>
</tbody>
</table>

Evaluation on test data (135 items):

<table>
<thead>
<tr>
<th>Before Pruning</th>
<th>After Pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size Errors</td>
<td>Size Errors</td>
</tr>
<tr>
<td>25 7 (5.2%)</td>
<td>7 4 (3.0%)</td>
</tr>
</tbody>
</table>

Figure 6.2: A pruned decision tree with its estimated error rate

pruned tree has higher error rate on the training data, it outperforms the original tree on the test data set which is unseen before.

6.3 Converting trees to rules

The previous section presents how to prune a decision tree so that it is both simpler and more accurate. Although the simplified trees are more concise and compact than the
physician fee freeze = n:
  | adoption of the budget resolution = y: democrat (151.0)
  | adoption of the budget resolution = u: democrat (1.0)
  | adoption of the budget resolution = n:
  |   | education spending = n: democrat (6.0)
  |   | education spending = y: democrat (9.0)
  |   | education spending = u: republican (1.0)

Figure 6.3: Portion of decision tree presented in Figure 6.1

originals, they are still awkward and complex to use. Large decision trees are difficult to
understand because each node has a specific context established by the results of tests
performed at its ancestor nodes. Figure 6.3 reprints the top portion of the decision tree
presented in Figure 6.1. Notice the last leaf in Figure 6.3 tests education spending,
giving class republican if the answer is unknown. It does not suggest that the test of
education spending among the requested detection is sufficient to decide whether it is
republican or democrat. The test of education spending makes sense only when read
in conjunction with the earlier tests conducted at its ancestor nodes. When we have
a large tree it can be very difficult to keep track of the continually changing context.

In a decision tree, the conditions that must be satisfied when a case is classified
by a leaf can be found by tracing all the test outcomes along the path from the root
to that leaf. In the tree of Figure 6.3, the only republican leaf is associated with
the test results of: physician fee freeze=n, adoption of the budget resolution=n,
and education spending=u. We can thus write the following rule:

```
if  physician fee freeze=n
    adoption of the budget resolution=n
    education spending=u
then  class republican
```

with the understanding that the conditions making up the rule antecedent are to be
interpreted as a conjunction. Therefore a rule covers a case if the case satisfies the rule's
antecedent conditions.

We can convert a decision tree to a collection of rules according to the aforementioned
method, one for each leaf in the tree. This effort could clarify the context of each leaf
node, however, it does not result in anything much simpler that the original tree. We

40
Rule 1:
physician fee freeze = n
-> class democrat [98.4%]

Rule 2:
synfuels corporation cutback = y
duty free exports = y
-> class democrat [97.5%]

Rule 3:
water project cost sharing = y
physician fee freeze = u
-> class democrat [70.7%]

Rule 4:
physician fee freeze = y
-> class republican [88.7%]

Rule 5:
physician fee freeze = u
mx missile = u
-> class republican [50.0%]

Default class: democrat

Figure 6.4: Rule set of decision tree presented in Figure 6.1

look forward to further compacting the rule set without affecting its accuracy. Let rule $R$ be: if $A$ then class $C$. Deleting one condition $X$ from the conditions $A$, we obtain a less rigid rule $R^-$: if $A^-$ then class $C$. Recall that in section 6.2, we present how C4.5 prune decision trees based on estimated pessimistic error rate. The same technique is used for simplify rule sets. If the pessimistic error rate of rule $R^-$ is less or equal than that of the original rule $R$, then condition $X$ can be deleted right away. Instead of looking at all possible subsets of conditions that could be deleted, C4.5 adopts a more efficient simplifying method: the condition that can be removed to produce the lowest pessimistic error rate of the generalized rule is always truncated first. This method is applied to the whole rule set exhaustively until each rule is reduced to its most concise format that produces the minimum pessimistic error rate.
We present the corresponding rule set produced by the same example that derives the decision trees of Figure 6.1 in Figure 6.4. The confidence percentage in the square brackets are calculated by the final estimated pessimistic error rate of the corresponding rule. The last rule in the rule set is the default class, which is also called default rule. This class is chosen if a case is not matched by any of the rules presented in the whole rule set. Therefore, C4.5 selects as the default that class which contains the most training cases not covered by any rule.

6.4 Options to enhance the predictive model

One of the key criteria to evaluate a predictive model is its predictability. In the last two sections, we describe the techniques used by C4.5 to simplify the predictive models without increasing the error rate. In this section, we introduce some options that can be applied, together with the default C4.5 predictive model building process, to reduce the estimated error rate for unseen cases.

6.4.1 Windowing option

A subset of the training cases called a window is selected randomly to grow a decision tree. This decision tree is then used to classify the training cases that have not been included in the window. Of course some of these cases are misclassified. A selection of these misclassifications was then added to the original window. Another tree is constructed from the expanded learning set. This cycle is repeated until a decision tree built from the current window correctly classifies all the training cases outside the window.

Windowing technology has two advantages. First, it produces more accurate trees. In order to produce better initial trees when the distribution of classes in the learning set is very unbalanced, C4.5 makes the distribution of classes in the initial window as uniformly as possible. Furthermore, with the same training set, C4.5 can select different sets of initial learning cases into the first window to produce different final trees. This provides two attractive alternatives:

- Select the tree with the lowest predicted error rate from all the trees that produced; and

4 which is the case in this research
Generalize rule sets from all the candidate trees and then composite a final rule set from all of them.

Empirical validations show that the final predictive rule set obtained in this way is more accurate than one obtained from a single tree[26].

The second benefit of using the windowing technology is faster construction of predictive models. C4.5 adds at least half of the exceptional cases to the current learning set at each cycle to facilitate the construction of a more accurate predictive model in the next window. C4.5 also stops before the tree correctly classifies all cases outside the window if it appears that the predictability of trees is not getting higher. Windowing can prevent unnecessary tree growth when the learning set contains some noisy data.

### 6.4.2 Grouping option

If we want to simplify the predictive model constructed from a multivalued attribute, one or more outcomes must be associated with a group of attribute values rather than a single value. Collections of attribute values associated with a single branch will be referred to as value groups. They are not to be confused with subsets of training cases. In some domains, appropriate groups of attribute values can be determined from domain knowledge. Consider the case of an attribute denoting a patient, we can have the following grouping methods:

- Grouped by age, each patient is either an infant, a child, an adult, or a senior.
- Some patients are smokers, some are not.
- Patients can be either fully covered or partially covered by their medical insurance plan.

Any or all of these groupings of the patients may be relevant to the classification task at hand. Where such well-established groupings are known beforehand, it makes sense to provide this information to the system by way of additional attributes, one for each potentially relevant grouping. So, in this case, we might add a multivalued attribute to give his/her age group, another true-false attribute to indicate whether the original attribute is a smoker, another four-valued attribute to indicate his/her age group, and so on.
6.4.3 Confidence factor

The confidence factor (CF) value affects decision tree pruning. The smaller the CF value the heavier the pruning goes. If the actual error rate of pruned trees on test cases is much higher than the estimated error rate, it is a good time to lower the CF value for heavier pruning.

6.4.4 Weight option

Sometimes, near-trivial tests can yield training cases having almost the same outcome. This can lead to unusual trees with hardly any predictive power. To avoid classification being affected by noise or indeterminacy, C4.5 requires that any test used in the tree must have at least two outcomes with a minimum number of cases. The default minimum for the weight factor is therefore 2. When there is a lot of noisy data a higher value may be applied.

6.5 Building the best predictive models

So far we present how C4.5 produces decision trees from training cases, the procedure of simplifying the original decision trees and the process of generating concise rule sets without compromising predictability. We also briefly introduce four options to run with the C4.5 to further reduce the predicted error rate and to manipulate the degree of pruning. The predictability and the size of the final models vary with the different options chosen to run with the C4.5. We are only interested in the models with the lowest estimated predictive error rate. In order to build such models, we have to conduct massive tests with varied options on the training data obtained for each hypothesis. The options listed in Table 6.1 are to be tested.

For test conducted with each option, we are interested in the results of both trees and rules. As for the decision trees we look into the following measures:

- **Size**: this is an index of the complexity of the decision trees.
  - before: number of branches counted before pruning.
  - after: number of branches counted after pruning
- **Error rate**: percentage of cases classified wrongly by the predictive model during various stages of building the decision trees. Recall that in section 6.2 we introduce the *cross-validation* technique that divides the available data into *N* blocks and then builds *N* different decision trees in each of which one block is omitted from the training data, and the resulting classification model is tested on the omitted block of cases. Therefore, we have the following three kinds of errors:

  - *Train set error rate*: is the average error rate of the training sets.
  - *Unseen set error rate*: is the average error rate of the test sets.
  - *Predicted error rate*: is the *estimated pessimistic error rate* which is explained in section 6.2.

As for the rule sets we are interested in two error rates which are:

- **Error rate**: is the average error rate of the training sets.
- **Unseen error rate**: is the average error rate of the test sets.

<table>
<thead>
<tr>
<th>No.</th>
<th>Symbol</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>default</td>
<td>with no option at all</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>group option. This causes the values of discrete attributes to be grouped for tests</td>
</tr>
<tr>
<td>3</td>
<td>m5</td>
<td>weight option. Discard cases that have less than 5 occurrences</td>
</tr>
<tr>
<td>4</td>
<td>m10</td>
<td>another weight option</td>
</tr>
<tr>
<td>5</td>
<td>m15</td>
<td>another weight option</td>
</tr>
<tr>
<td>6</td>
<td>c15</td>
<td>pruning option. The default pruning factor is 25% therefore this option goes for a heavier pruning</td>
</tr>
<tr>
<td>7</td>
<td>c10</td>
<td>even more heavier pruning at 10%</td>
</tr>
<tr>
<td>8</td>
<td>c15 m5</td>
<td>this is a combination of pruning and weight options run at the same time</td>
</tr>
<tr>
<td>9</td>
<td>t10</td>
<td>windowing option. 10 trees will be grown before the best is chosen</td>
</tr>
<tr>
<td>10</td>
<td>t15</td>
<td>windowing option with 15 trees</td>
</tr>
<tr>
<td>11</td>
<td>t5</td>
<td>windowing option with 5 trees</td>
</tr>
<tr>
<td>12</td>
<td>t5 c15</td>
<td>combination of windowing option and pruning option</td>
</tr>
<tr>
<td>13</td>
<td>c15 c10</td>
<td>another combination of windowing and pruning factors</td>
</tr>
</tbody>
</table>

Table 6.1: Options run with C4.5 for selecting the best predictive model
Table 6.2 help us to clarify what we are looking for in order to select the best predictive models. The rows are the options presented in Table 6.1 and the columns are the criteria for selecting the best predictive model. In order to fill in this table we need to run data collected from each hypothesis with thirteen different options. With three hypotheses and four predictive models\(^5\), there are fifty-two rounds of C4.5 tree building and C4.5 rule generating processes.

<table>
<thead>
<tr>
<th>No.</th>
<th>Options</th>
<th>Decision Trees</th>
<th></th>
<th></th>
<th></th>
<th>Rules</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Size before</td>
<td>after</td>
<td>Error rate</td>
<td>Train set error rate</td>
<td>Unseen error rate</td>
<td>Predicted error rate</td>
<td>Train set error rate</td>
</tr>
<tr>
<td>1</td>
<td>default</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>m5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>m10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>m15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>c15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>c15 m5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>t10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>t15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>t5 c15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>t15 c10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: A composition of criteria for selecting the best predictive models

### 6.6 Data preparation

Before filling data into Table 6.2 for each model, we describe how to prepare input data for the C4.5 algorithm. Each case in a training set that are examined by C4.5 consists of two types of variables: a set of independent variables and only one dependent variable.  

\(^5\)we verify hypothesis 2 with design metrics and code metrics separately
6.6.1 Dependent variable vs. Independent variables

*Independent* variables are those that are manipulated, whereas *dependent* variables are only measured or registered. This distinction appears terminologically confusing based on the fact that all variables depend on something. The terms *dependent* and *independent* variable apply mostly to experimental research where some variables are manipulated, and in this sense they are independent from the initial reaction patterns, features, intentions, etc. of the subjects. Some other variables are expected to be dependent on the manipulation or experimental conditions. That is to say, they depend on what the subject will do in response.

In this research, the dependent variable is the empirical measure for reusability. Different aspects can be considered to measure empirically the reusability of a component depending on the adopted point of view. One aspect is the amount of work needed to reuse a component from a version of a system to another version of the same system. Another aspect is the amount of work needed to reuse a component from a system to another system of the same domain. This latter aspect was adopted as the empirical reusability measure for our experiment. To define the possible values for this measure, we worked with a team in CRIM⁶ specializing in developing intelligent multiagents systems⁷. The obtained values classes are:

1. *Totally reusable*: means that the component is generic to a certain domain (in our case *intelligent multiagents systems*).

2. *Reusable with minimum rework*: means that less than 25% of the code needs to be altered to reuse the component in a new system of the same domain.

3. *Reusable with high amount of rework*: means that more than 25% of the code needs to be changed before reusing the component in a new system of the same domain.

4. *Not reusable at all*: means that the component is too specific to be reused.

For the ease of input files and readability, we simply enumerate the above reusability measures. Therefore, the reusability of a class is a value chosen from the list \{1, 2, 3, or 4\} depend on how reusable it is. The higher the reusability the lower this value becomes. For instance, a class that is *totally reusable* is given reusability 1 meanwhile a class that is *not reusable at all* is ranked as reusability 4.

⁶Centre de recherche informatique de Montreal, Montreal, Canada
⁷Details on the work of this team can be found in http://www.crim.ca/sbc/english/lalo/
The independent variables are different sets of metrics values that are extracted in section 5.3. Therefore we have four sets of independent variables, which are metrics values, to be paired with one set of dependent variables, which are reusability measures. Table 6.3 gives an example of a portion of input data for hypothesis 1 before converted to C4.5 input files.

<table>
<thead>
<tr>
<th>No.</th>
<th>Class name</th>
<th>Independent variables</th>
<th>Dependent variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Action</td>
<td>0 0 0 0 0 7 0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>ActionList</td>
<td>0 0 0 0 0 7 0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Agenda</td>
<td>0 0 0 0 0 12 2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>ArgsSet</td>
<td>0 0 0 0 0 15 0</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>BasicAgent</td>
<td>0 2 0 1 2 31 12</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Belief</td>
<td>0 0 0 0 0 19 0</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 6.3: Dependent and independent variables

### 6.6.2 The C4.5 input files

C4.5 has its unique way of input class and attributes, i.e. dependent variables and independent variables. The most important file is the names file that provides names for classes, attributes and the types of their attribute values. The names files consists of a series of entries. The first entry in the names file gives the class names, separated by commas and ended with a period. There must be at least two class names and their order is not important. The rest of the file consists of a single entry for each attribute. An attribute entry begins with the attribute name followed by a colon, and then a specification of the values that the attribute can take. C4.5 allows four specifications:

- **ignore**: means the value of the attribute to be disregarded.
- **continuous**: indicates that the attribute has either integer or floating point values.
- **discrete N**: specifies that the attribute has discrete values, and there are no more than N of them. N is a positive integer.
- enumerate: a list of names separated by commas. This is like the enumerated types in C++.

We choose continuous type for all attributes because all extracted metrics values are in the form of either integer or float. Figure 6.5 helps to illustrate the proper content of a name file. The first line contains the class name. In our case, it is the enumerated reusability values. The next seven entries, one for each chosen metrics to verify hypothesis 1, consists of an attribute name followed by a colon and then its type, which is continuous. The vertical bar | starts the comments to increase the readability of the file.

```
1,2,3,4. | reusability
```

<table>
<thead>
<tr>
<th>DIT: continuous.</th>
<th>Depth Of Inheritance Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIT: continuous.</td>
<td>Height Of Inheritance Tree</td>
</tr>
<tr>
<td>NOA: continuous.</td>
<td>Number Of Ancestors</td>
</tr>
<tr>
<td>NOC: continuous.</td>
<td>Number Of direct Child</td>
</tr>
<tr>
<td>NOD: continuous.</td>
<td>number of Decendants</td>
</tr>
<tr>
<td>OIM: continuous.</td>
<td>number of inherited methods</td>
</tr>
<tr>
<td>NOP: continuous.</td>
<td>number of polymorphic methods</td>
</tr>
</tbody>
</table>

Figure 6.5: C4.5 name file that defines reuse classification and attributes for hypothesis 1

```
0,0,0,0,0,7,0,4
0,0,0,0,0,7,0,4
0,0,0,0,0,12,2,2
0,0,0,0,0,15,0,4
0,2,0,1,2,31,12,4
0,0,0,0,0,19,0,4
```

Figure 6.6: A portion of C4.5 data file for hypothesis 1

The data file is used to describe the training cases from which decision trees and rule sets are to be constructed. Each line describes one case, providing the values for all the attributes, i.e. metrics values, and then the case’s class, i.e. reusability. Values are separated by commas and each case is terminated by a period. The attribute values must appear in the same order that the attributes were given in the names file. Figure 6.6 gives a portion of the data file corresponding to the name file in Figure 6.5.
6.7 Selection of the model building options

Before presenting the actual predictive models in the next chapter, we would like to select the best model building options for each hypothesis. This is done by observing the complexities and error rates filled in the format proposed in Table 6.2. Table 6.4, 6.5, 6.6 and 6.7 records data for hypothesis 1, hypothesis 2 verified by design metrics, hypothesis 2 verified by code metrics and hypothesis 3, respectively. The minimum error rate in each column is italicized for the purpose of comparison and selection.

For the models produced for hypothesis 1 (Table 6.4), the $t_{15}^8$ option obviously produced the most attractive predictive models. It has the lowest train set error rate, unseen error rate and predicted error rate for the decision trees. It also holds the lowest unseen error rate for the rule sets. Although its train set error rate which is 25.5 of the rule models is not the smallest, it is very close to the lowest error rate in that category which is 25.3. The group option $m$ does not help in building better models. The weight options $m[n]$ do enhance the predictability but their results are still not as good as those of the windowing technology. Models produced by heavier tree pruning options $c[n]$ reduce the error rate in some limited way. However they do not produce the best models. The combinations of the windowing option and the pruning options $t[n]c[m]$ weaken the power of the former. There is no doubt that we should choose $t_{15}$ as the best option to build the predictive model for hypothesis 1.

The error rates from models generated for hypothesis 2 (design metrics) in Table 6.5 has a very similar pattern to hypothesis 1. Windowing option $t_{15}$ outperforms other options in almost every column. Although the unseen error rate 47.5 of the rule models is not the lowest it is the third smallest in the entire column and very close to the minimum 45.7. We therefore select $t_{15}$ to produce the predictive model for hypothesis 2 verified by design metrics.

Options for hypothesis 2 (code metrics) is hard to choose. Table 6.6 does not has the same look as the previous two tables have. The minimum unseen error rate for both the tree models and the rule models are surprisingly produced by the weight option $m_{10}$. However, we are convinced by the fact that windowing option $t_{15}$ generates the lowest predicted error rate and two other error rates among all five different error rates we compare. Thus $t_{15}$ is selected.

Option $t_{15}$ is chosen for hypothesis 3 by observing Table 6.7 which is consistent with the first two tables.

---

^{8}$windowing option with the initial window size of 15
<table>
<thead>
<tr>
<th>No.</th>
<th>Options</th>
<th>Decision Trees</th>
<th></th>
<th>Rules</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size</td>
<td>Error rate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>before</td>
<td>after</td>
<td>Train set error rate</td>
<td>Unseen error rate</td>
</tr>
<tr>
<td>1</td>
<td>default</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>3</td>
<td>m5</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>4</td>
<td>m10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>5</td>
<td>m15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>6</td>
<td>c15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>7</td>
<td>c10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>8</td>
<td>c15 m5</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>9</td>
<td>t10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>10</td>
<td>t15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>11</td>
<td>t5</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>12</td>
<td>t5 c15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>13</td>
<td>t15 c10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
</tbody>
</table>

Table 6.4: Complexities and error rates of models for hypothesis 1

<table>
<thead>
<tr>
<th>No.</th>
<th>Options</th>
<th>Decision Trees</th>
<th></th>
<th>Rules</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Size</td>
<td>Error rate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>before</td>
<td>after</td>
<td>Train set error rate</td>
<td>Unseen error rate</td>
</tr>
<tr>
<td>1</td>
<td>default</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>3</td>
<td>m5</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>4</td>
<td>m10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>5</td>
<td>m15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>6</td>
<td>c15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>7</td>
<td>c10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>8</td>
<td>c15 m5</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>9</td>
<td>t10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>10</td>
<td>t15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>11</td>
<td>t5</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>12</td>
<td>t5 c15</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
<tr>
<td>13</td>
<td>t15 c10</td>
<td>35</td>
<td>23.6</td>
<td>17.9</td>
<td>52.4</td>
</tr>
</tbody>
</table>

Table 6.5: Complexities and error rates of models for hypothesis 2 (design metrics)
<table>
<thead>
<tr>
<th>No.</th>
<th>Options</th>
<th>Decision Trees</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Size</td>
<td>Error rate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>before</td>
<td>after</td>
</tr>
<tr>
<td>1</td>
<td>default</td>
<td>30.8</td>
<td>23.8</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>30.8</td>
<td>23.8</td>
</tr>
<tr>
<td>3</td>
<td>m5</td>
<td>14</td>
<td>10.4</td>
</tr>
<tr>
<td>4</td>
<td>m10</td>
<td>7.6</td>
<td>6.8</td>
</tr>
<tr>
<td>5</td>
<td>m15</td>
<td>5.2</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>c15</td>
<td>30.8</td>
<td>20.4</td>
</tr>
<tr>
<td>7</td>
<td>c10</td>
<td>30.8</td>
<td>20.2</td>
</tr>
<tr>
<td>8</td>
<td>c15 m5</td>
<td>14</td>
<td>10.4</td>
</tr>
<tr>
<td>9</td>
<td>t10</td>
<td>31.2</td>
<td>23.6</td>
</tr>
<tr>
<td>10</td>
<td>t15</td>
<td>30.8</td>
<td>22.2</td>
</tr>
<tr>
<td>11</td>
<td>t5</td>
<td>29.8</td>
<td>21.4</td>
</tr>
<tr>
<td>12</td>
<td>t5 c15</td>
<td>29.8</td>
<td>21.4</td>
</tr>
<tr>
<td>13</td>
<td>t15 c10</td>
<td>29.4</td>
<td>18.4</td>
</tr>
</tbody>
</table>

Table 6.6: Complexities and error rates of models for hypothesis 2 (code metrics)

<table>
<thead>
<tr>
<th>No.</th>
<th>Options</th>
<th>Decision Trees</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Size</td>
<td>Error rate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>before</td>
<td>after</td>
</tr>
<tr>
<td>1</td>
<td>default</td>
<td>30.8</td>
<td>28.6</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>30.8</td>
<td>28.6</td>
</tr>
<tr>
<td>3</td>
<td>m5</td>
<td>16.4</td>
<td>10.4</td>
</tr>
<tr>
<td>4</td>
<td>m10</td>
<td>5.8</td>
<td>5.4</td>
</tr>
<tr>
<td>5</td>
<td>m15</td>
<td>5.2</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>c15</td>
<td>30.8</td>
<td>23.6</td>
</tr>
<tr>
<td>7</td>
<td>c10</td>
<td>30.8</td>
<td>23.2</td>
</tr>
<tr>
<td>8</td>
<td>c15 m5</td>
<td>16.4</td>
<td>10.4</td>
</tr>
<tr>
<td>9</td>
<td>t10</td>
<td>31.8</td>
<td>29.2</td>
</tr>
<tr>
<td>10</td>
<td>t15</td>
<td>31.8</td>
<td>29.2</td>
</tr>
<tr>
<td>11</td>
<td>t5</td>
<td>32.8</td>
<td>28</td>
</tr>
<tr>
<td>12</td>
<td>t5 c15</td>
<td>32.8</td>
<td>24.6</td>
</tr>
<tr>
<td>13</td>
<td>t15 c10</td>
<td>31.8</td>
<td>24.4</td>
</tr>
</tbody>
</table>

Table 6.7: Complexities and error rates of models for hypothesis 3
Chapter 7

Experimental Results and Validations

We present the predictive models generated by C4.5 for all three hypothesis in this chapter. Each model is discussed briefly from the point of selected metrics and their thresholds. We then validate the predictive models using both the training data and the testing data.

7.1 The predictive models

For each hypothesis, C4.5 induces a rule-based predictive model. Most of the selected metrics in each generated model confirm the intuitive hypothesis we have stated about reusability. It is for example the case for Depth of Inheritance Tree (DIT), Number of Children (NOC), and Number of Polymorphic methods (NOP) concerning complexity vs. Reusability hypothesis. It is also the case for Class Size in Bytes (CSB) and Number of Trivial methods (NOT) concerning complexity vs. Reusability hypothesis. However, let us present a more qualitative interpretation of the models generated by C4.5, given each time examples of relevant induced rules.

7.1.1 Hypothesis 1

Each one of the following rules of the predictive model of hypothesis 1 describes a situation in terms of inheritance metrics, where a class can be considered as reusable or not. In the reminder of this section we will comment three of them.
<table>
<thead>
<tr>
<th>Rule 1:</th>
<th>Rule 2:</th>
<th>Rule 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOC &gt; 2</td>
<td>DIT &gt; 1</td>
<td>DIT &gt; 0</td>
</tr>
<tr>
<td>NOP &gt; 4</td>
<td>NOP &gt; 6</td>
<td>NOC &lt;= 2</td>
</tr>
<tr>
<td>NOP &lt;= 6</td>
<td>→ class 2 [63.0%]</td>
<td>OIM &lt;= 14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOP &gt; 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOP &lt;= 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→ class 3 [75.8%]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 4:</th>
<th>Rule 5:</th>
<th>Rule 6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIT &gt; 0</td>
<td>DIT &lt;= 0</td>
<td>DIT &gt; 0</td>
</tr>
<tr>
<td>OIM &lt;= 21</td>
<td>HIT &lt;= 0</td>
<td>HIT &lt;= 0</td>
</tr>
<tr>
<td>NOP &lt;= 4</td>
<td>OIM &lt;= 11</td>
<td>NOP &lt;= 4</td>
</tr>
<tr>
<td>→ class 3 [66.2%]</td>
<td>NOP &gt; 1</td>
<td>→ class 4 [73.0%]</td>
</tr>
<tr>
<td></td>
<td>→ class 3 [66.2%]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 7:</th>
<th>Rule 8:</th>
<th>Rule 9:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIT &gt; 2</td>
<td>NOC &lt;= 2</td>
<td>NOP &gt; 10</td>
</tr>
<tr>
<td></td>
<td>NOP &lt;= 1</td>
<td>→ class 4 [50.0%]</td>
</tr>
<tr>
<td></td>
<td>→ class 4 [63.3%]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default rule:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>→ class 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.1: Predictive model for hypothesis 1 in rule based model

The rule 2 presents the impact of the combination of the depth of inheritance with number of virtual methods on the reusability. A class with DIT value greater than one is not in the top portion of its hierarchy. It may not be good for reuse in the sense that it somehow sufficiently specific. However, if that class contains sufficient number of virtual methods (NOP > 6 in the rule compared to the average NOP = 4 for the whole system), it can be considered as a generic class because there is part of its behavior that can be specified and extended by its descendant classes. Such classes are reusable in the same domain without major rework (level 2). In the rule 8, it is stated that a class with a small number of children and with almost no polymorphic method is too specific to be reusable. Therefore reuse this kind of classes is strongly discouraged. Due to the small size of our learning set, some rules can be considered as noise. It is the case of rule 9 where a class with a relatively big number of virtual methods is considered as too specific to be reusable. Finally, we can notice that the metric NOA (the number of distinct ancestors) is never used in the rules. In LALO, multiple inheritance is not used. Thus, NOA is equal to DIT so it not used because it is redundant.
7.1.2 Hypothesis 2 (code metrics)

With the code coupling metrics, C4.5 produces the following reusability predictive model. Two rules of this model are explained in the reminder of this section.

The rule 1 shows that a class with extremely high Stamp-Reference Control Interconnection export measure is good for reuse (average value is 2 for the system). This can be explained by its usefulness and ease of use of the class. The more other classes to which a class exports its service, the more general this class is. The export interdependencies somehow encourage reuse. With respect to rule 6, a class with considerable amount of import coupling (average value is 0.5) with its environment is bad for reuse. Import couplings make a class depend on other classes that are not desirable which complicates the process of extraction and increase the risk of being more fault-prone.

<table>
<thead>
<tr>
<th>Rule 1:</th>
<th>Rule 2:</th>
<th>Rule 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{StC}1\text{exp} &gt; 7 )</td>
<td>( \text{RD}1\text{exp} &gt; 8 )</td>
<td>( \text{NPIexp} &lt;= 1 )</td>
</tr>
<tr>
<td>( \rightarrow \text{class 1} ) [63.0%]</td>
<td>( \text{SR}1\text{exp} &gt; 2 )</td>
<td>( \text{RD}1\text{exp} &gt; 1 )</td>
</tr>
<tr>
<td>( \rightarrow \text{class 2} ) [74.0%]</td>
<td>( \rightarrow \text{class 1} ) [50.0%]</td>
<td>( \text{RD}1\text{exp} &lt;= 8 )</td>
</tr>
<tr>
<td>( \rightarrow \text{class 4} ) [89.9%]</td>
<td>( \rightarrow \text{class 4} ) [88.2%]</td>
<td>( \rightarrow \text{class 4} ) [91.2%]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 4:</th>
<th>Rule 5:</th>
<th>Rule 6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{RCIimp} &gt; 0 )</td>
<td>( \text{StRCIexp} &gt; 0 )</td>
<td>( \text{SDIimp} &gt; 1 )</td>
</tr>
<tr>
<td>( \text{RCIexp} &lt;= 5 )</td>
<td>( \text{TYIexp} &lt;= 6 )</td>
<td>( \rightarrow \text{class 4} ) [84.1%]</td>
</tr>
<tr>
<td>( \text{SRDIimp} &lt;= 1 )</td>
<td>( \rightarrow \text{class 4} ) [89.9%]</td>
<td>( \rightarrow \text{class 4} ) [88.2%]</td>
</tr>
<tr>
<td>( \text{SrcRCIexp} &lt;= 6 )</td>
<td>( \rightarrow \text{class 2} ) [74.0%]</td>
<td>( \rightarrow \text{class 3} ) [69.4%]</td>
</tr>
<tr>
<td>( \text{TYIexp} &lt;= 5 )</td>
<td>( \rightarrow \text{class 3} ) [69.4%]</td>
<td>( \rightarrow \text{class 3} )</td>
</tr>
</tbody>
</table>

Figure 7.2: Predictive model for hypothesis 2 (code metrics) in rule based model
7.1.3 Hypothesis 2 (design metrics)

The following predictive model describes situations where a class is considered as reusable or not with respect to design coupling metrics. Two of them are explained in the reminder of this section.

Rule 4 reconfirms with what we found with code metrics. A class can be reused if its export coupling is much higher than average (average OCAEC is 1.75 in the system) and it does not heavily depend on other class (average ACMID is 0.90 in the system). As for hypothesis 1, some rules like rule 3 are consequence of noise. Indeed, a class that needs a considerable number of services from of its environment is hard to reuse.

<table>
<thead>
<tr>
<th>Rule 1:</th>
<th>Rule 2:</th>
<th>Rule 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ACMIC} &gt; 2$</td>
<td>$\text{OCAEC} &gt; 2$</td>
<td>$\text{ACMIC} &gt; 4$</td>
</tr>
<tr>
<td>$\text{ACMIC} \leq 4$</td>
<td>$\text{OCAEC} &gt; 4$</td>
<td>$\rightarrow \text{class 1 [50.0%]}$</td>
</tr>
<tr>
<td>$\rightarrow \text{class 1 [54.6%]}$</td>
<td>$\text{ACMIC} \leq 4$</td>
<td>$\rightarrow \text{class 2 [50.0%]}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 4:</th>
<th>Rule 5:</th>
<th>Rule 6:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{OCAEC} &gt; 4$</td>
<td>$\text{OCAEC} \leq 2$</td>
<td>$\text{ACMIC} \leq 0$</td>
</tr>
<tr>
<td>$\text{ACMIC} \leq 2$</td>
<td>$\text{DCMEC} &gt; 0$</td>
<td>$\rightarrow \text{OMMIC} \leq 20$</td>
</tr>
<tr>
<td>$\rightarrow \text{class 2 [50.0%]}$</td>
<td>$\rightarrow \text{class 2 [45.3%]}$</td>
<td>$\rightarrow \text{OMMIEC} \leq 17$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 7:</th>
<th>Rule 8:</th>
<th>Rule 9:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{OCAEC} \leq 1$</td>
<td>$\text{ACMIC} \leq 9$</td>
<td>$\text{ACMIC} \leq 2$</td>
</tr>
<tr>
<td>$\text{OMMIC} \leq 9$</td>
<td>$\text{ACMIC} &gt; 0$</td>
<td>$\rightarrow \text{OMMEC} &gt; 3$</td>
</tr>
<tr>
<td>$\rightarrow \text{OMMIC} \leq 20$</td>
<td>$\rightarrow \text{OMMEC} \leq 17$</td>
<td>$\rightarrow \text{class 4 [72.6%]}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 10:</th>
<th>Default rule:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{OMMEC} &gt; 0$</td>
<td>$\rightarrow \text{class 3}$</td>
</tr>
<tr>
<td>$\text{OMMEC} \leq 1$</td>
<td>$\rightarrow \text{class 4 [61.2%]}$</td>
</tr>
</tbody>
</table>

Figure 7.3: Predictive model for hypothesis 2 (design metrics) in rule based model

56
7.1.4 Hypothesis 3

The following rules of the predictive model of hypothesis 3 describe situations where a class can be considered as reusable or not using the chosen complexity metrics. Two of them are explained in details.

According to rule 5, a class that defines no abstract data type (average NAD is 4.2 in the system) and that has a very small set of attributes (CSB reflects the size of all attributes declared in a class, its average value is 127 in the system) can be considered as a simple class. But if a component is too simple it may not be worth reusing because the combined costs of extraction, retrieval, and integration exceed its intrinsic value, making reuse very impractical[10]. Fortunately, the value of NOT greater than two (average NOT

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT &lt;= 4</td>
<td>RFC &gt; 52</td>
<td>WMC &lt;= 9</td>
</tr>
<tr>
<td>NOP &gt; 4</td>
<td>NRA &lt;= 2</td>
<td>CSB &lt;= 12</td>
</tr>
<tr>
<td>NPM &lt;= 0.63</td>
<td>NOO &lt;= 6</td>
<td>→ class 2 [63.0%]</td>
</tr>
<tr>
<td>NPM &lt;= 0.77</td>
<td>NPM &lt; 1.23</td>
<td>Rule 3:</td>
</tr>
<tr>
<td>CSB &lt;= 52</td>
<td>→ class 2 [70.7%]</td>
<td></td>
</tr>
<tr>
<td>→ class 2 [75.8%]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 4</th>
<th>Rule 5</th>
<th>Rule 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC &lt;= 19</td>
<td>NOT &gt; 2</td>
<td>NOT &gt; 12</td>
</tr>
<tr>
<td>NPA &gt; 0</td>
<td>NAD &lt;= 0</td>
<td>NOO &gt; 6</td>
</tr>
<tr>
<td>NOO &lt;= 3</td>
<td>CSB &lt;= 12</td>
<td>→ class 1 [50.0%]</td>
</tr>
<tr>
<td>CSB &gt; 12</td>
<td>→ class 2 [54.6%]</td>
<td></td>
</tr>
<tr>
<td>→ class 2 [50.0%]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 7</th>
<th>Rule 8</th>
<th>Rule 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT &gt; 0</td>
<td>CIS &lt;= 23</td>
<td>RFC &gt; 88</td>
</tr>
<tr>
<td>NOP &lt;= 0</td>
<td>NPM &gt; 0.77</td>
<td>→ class 4 [70.7%]</td>
</tr>
<tr>
<td>NPA &lt;= 0</td>
<td>CSB &gt; 4</td>
<td></td>
</tr>
<tr>
<td>CSB &gt; 12</td>
<td>→ class 4 [78.4%]</td>
<td></td>
</tr>
<tr>
<td>→ class 4 [89.1%]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule 10</th>
<th>Default rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPM &lt;= 0.77</td>
<td>→ class 4</td>
</tr>
<tr>
<td>CSB &gt; 52</td>
<td>Default rule:</td>
</tr>
<tr>
<td>→ class 3 [52.8%]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.4: Predictive model for hypothesis 3 in rule based model
is 4.2 in the system) somehow sets a threshold for those simple but useful classes that deserve reuse. In the rule 9, RFC metric indicates how big the set of methods that can potentially be executed in response to a message received by an object of a class[12]. The Response Set for a class encompasses the set of all methods defined in the current class and all methods that can be called by each locally defined method. The larger the RFC value the greater the complexity of the class. Furthermore, compared to NOM and WMC which do not reflect any interdependencies, high measure in RFC metrics also indicates the potential communication between the class and other systems[12] which makes reuse more sophisticated.

7.2 Validation of the predictive models

Analyzing the metrics and their thresholds can provide some information about how close the proposed suites of metrics are related to reusability. However we need a more scientific evaluation method to validate these predictive models. In this section, we first propose a validation method and then present the results obtained from both the training set and the testing set using the proposed method.

7.2.1 Validation method

To evaluate the class reusability characterization model based on our measures, we need criteria for evaluating the overall model accuracy. Evaluating model accuracy tells us how good the model is expected to be as a predictor. If the characterization model based on our suite of measures provides a good accuracy it means that our measures are useful in identifying reusable classes. Two criteria for evaluating the accuracy of predictions are the measures of correctness and completeness.

- Correctness: is defined as the percentage of C++ classes that were deemed as k-reusable and were actually k-reusable (k represents the level of reusability defined in section 6.6). We want to maximize correctness because if correctness is low, then the model is identifying more C++ classes as being k-reusable when they really are not k-reusable.

- Completeness: is defined as the percentage of those k-reusable C++ classes that were judged as k-reusable. We want also to maximize completeness because as completeness decreases, more C++ classes that were k-reusable are misidentified as not k-reusable.

58
Table 7.1: Four-class classification performance matrix

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>$n_{11}$</td>
<td>$n_{12}$</td>
</tr>
<tr>
<td>level 2</td>
<td>$n_{21}$</td>
<td>$n_{22}$</td>
</tr>
<tr>
<td>level 3</td>
<td>$n_{31}$</td>
<td>$n_{32}$</td>
</tr>
<tr>
<td>level 4</td>
<td>$n_{41}$</td>
<td>$n_{42}$</td>
</tr>
<tr>
<td>Correctness</td>
<td>%</td>
<td>$\frac{n_{11}}{\sum_{i=1}^{4} n_{1i}}$</td>
</tr>
</tbody>
</table>

Table 7.1 provides a intuitive description of how we obtain measures of completeness and correctness. Notice that the columns are reusability predicted by the C4.5 classification models and the rows are the original reuse measures given by experts. Finally, the accuracy of each model can be measured by the following formula:

\[
\text{Accuracy} = \frac{\sum_{i,j=1}^{4} n_{ij}}{\sum_{i=1}^{4} n_{i4}}
\]

It simply sums up the number of classes along the diagonal grids in Table 7.1 and then divided by the total number of classes in the whole data set.

### 7.2.2 Validation tool

A validation tool called *Reusability Consult* is developed in Visual C++ in order to systematically and accurately validate large amount of unseen testing cases. Please refer to Appendix A for detailed information regarding the functionality and the operation of this tool.

### 7.2.3 Validation results

Using the validation method proposed above, we validate the predictive models for all three hypotheses with the aid of *Reusability Consult*. The validations are done by applying
We first present the validation results of the training data in Table 7.2, 7.3, 7.4 and 7.5. The predictive model of hypothesis 3 (Table 7.5) produces the highest accuracy rate (89.3%) among all four tables. It has a very clear diagonal pattern which is formed by accurately predicted reusable classes. This fact is a strong support to hypothesis 3 which states that complexity affects reusability. The two models of hypothesis 2 (Table 7.3 and 7.4) demonstrate very similar behaviors. Their predictability are not as superb as of hypothesis 3. However, a closer look at these two tables reveals that almost all misclassified classes are next to the diagonal grids. Furthermore, the numbers of the misclassifications in the two models are very small compared to the total class size of the training set. This assures that coupling, measured from different aspects, relates to reusability. Hypothesis 1’s classification model generates the worst behavior. The misclassified classes are scattered all over the table. None of the classes of reusability 1 is predicted correctly. Two of the originally most reusable classes were classified as the least reusable class. As we study the class hierarchy of the system that produce the training data, we find that the system’s inheritance relationship does not form a inheritance tree with a decent level. Instead, its classes hierarchy appears to be rather flat. Therefore, the learning data dose not provide C4.5 with enough knowledge of the relationship between class inheritance and reusability. Nevertheless, the overall accuracy of the model (73.8%) is still acceptable.

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>level 2</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>level 3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>level 4</td>
<td>0</td>
<td>54.2</td>
</tr>
</tbody>
</table>

Accuracy = 73.8%

Table 7.2: Validate hypothesis 1 using training data

As we have discussed in the previous chapter, the error rates of a model when applied...
Table 7.3: Validate hypothesis 2 (design metrics) using training data

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>level 2</td>
<td>1</td>
<td>10.4</td>
</tr>
<tr>
<td>level 3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>level 4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Correctness</td>
<td>%</td>
<td>85.7</td>
</tr>
</tbody>
</table>

Accuracy = 88.1%

Table 7.4: Validate hypothesis 2 (code metrics) using training data

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>level 2</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>level 3</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>level 4</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>Correctness</td>
<td>%</td>
<td>100</td>
</tr>
</tbody>
</table>

Accuracy = 86.9%

Table 7.5: Validate hypothesis 3 using training data

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>level 2</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>level 3</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>level 4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Correctness</td>
<td>%</td>
<td>100</td>
</tr>
</tbody>
</table>

Accuracy = 89.3%

to the unseen cases provides a more trustworthy evaluation of its predictability. Three models are tested by unseen data sets. We are unfortunately not able to obtain the values of the code metrics needed for hypothesis 2. The extraction of the code metrics for such
a medium size system is so time-consuming\footnote{It took 4 people about a month to extract the learning data set} that we have to postpone this part of the validation to the future. However, let us present the results of the available verification results obtained from the unseen cases.

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>level 2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>level 3</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>level 4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Correctness</td>
<td>0.0</td>
<td>50.0</td>
</tr>
</tbody>
</table>

Accuracy = 59.5%

Table 7.6: Validate hypothesis 1 using testing data

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>level 2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>level 3</td>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>level 4</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Correctness</td>
<td>33.3</td>
<td>85.7</td>
</tr>
</tbody>
</table>

Accuracy = 67.5%

Table 7.7: Validate hypothesis 2 (design metrics) using testing data

<table>
<thead>
<tr>
<th>Real reusability</th>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>level 1</td>
<td>level 2</td>
</tr>
<tr>
<td>level 1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>level 2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>level 3</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>level 4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Correctness</td>
<td>28.6</td>
<td>50.0</td>
</tr>
</tbody>
</table>

Accuracy = 65.5%

Table 7.8: Validate hypothesis 3 using testing data
We are not surprised to see that the overall performance of the models tested on the unseen cases are worse than those of the training cases. Hypothesis 1 (Table 7.6) remains to be the least accurate model of the three. The error rate patterns are similar to the ones in Table 7.2 where data are scattered all over the table. It is because the learning data set were taken from a system that dose not reflect the traditional object-oriented hierarchy structure. Therefore, it is lack of predictability in such category. The model for hypothesis 2 (Table 7.7 is the best among the three. Although the overall accuracy is much lower than its counterpart (Table 7.3), all the misclassified cases are located right beside the diagonal grids. The performance of the model generated for hypothesis 3 (Table 7.8) is close to the one of hypothesis 2. Its data layout appears to be a bit more scattered. However, the misclassified classes are adjacent to the diagonal line and their percentages are relatively small.

We may not be fully convinced for the predictability of the models on the unseen cases. But let us rearrange the results of Table 7.6, 7.7, and 7.8 in a way so that we can see clearly from another point of view. Instead of classifying reusability in four different categories as we have proposed in section 6.6.1:

- Totally reusable,
- Reusable with minimum amount of rework,
- Reusable with high amount of rework, and
- Not reusable at all.

we group the first two categories together and name them as *easy* for reuse. The latter two are combined together so that they are referred as *hard* to reuse. This change alters the presentation of the validation results which are illustrated in the following three tables.

<table>
<thead>
<tr>
<th>Predicted reusability</th>
<th>Completeness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Easy</td>
</tr>
<tr>
<td>Real reusability</td>
<td>Easy</td>
</tr>
<tr>
<td></td>
<td>Hard</td>
</tr>
<tr>
<td>Correctness</td>
<td>%</td>
</tr>
<tr>
<td>Accuracy = 86%</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.9: Validate hypothesis 1 using testing data, altered presentation

‘ 63
Table 7.9, 7.10, and 7.11 actually magnify the diagonal grids of their predecessors. By grouping classes of similar reuse level we are able to see the usefulness of the predictive models produced for all three hypothesis.

### 7.3 Discussion

So far, we have described an experiment to verify three hypotheses of correlation between reusability and quantitative attributes of OO software components (inheritance, coupling, and complexity metrics). In the experiment, we adopt a machine learning approach using C4.5. The results of our experiment show that the selected metrics can predict with a high level of accuracy the classes that can be potentially reusable. The four predictive models obtained give very satisfying results for both the learning cases and the unseen testing cases.

We thus claim that all three hypotheses are correct with respect to the domain in which this research work is performed. Nevertheless, software engineering working on different domains will have different opinions on the reusability of a component. Some components are more reusable in one domain and less reusable in others, so they are
incomparable with respect to reusability. The aim of this research is not to search for a metric or a set of metrics that can magically predict reusability. Instead, our goal is to look for the specific properties or aspects of OO software components that are pertinent to reuse.

The predictive models we obtained are only applied to the specific team that develops the open multiagent system from which the rules are generated. Different groups may have different sets of rules, mostly when the dependent variable definition (i.e., reusability) differs. That is the most important characteristic of this method: it generates rules, and therefore guidelines, to the specific environment to which it is applied. This allows each software group to improve its performance by finding areas on which to focus its attention.
Chapter 8

Conclusion and Future Work

In this thesis, we explore the metrics based detection of reusable object-oriented software components using a machine learning algorithm. Our work is unique in its comprehensive study of the detection for reusable components. We also establish a methodology of finding properties of OO components that are closely related to reusability. This process can be repeated for refined results. In this chapter we first conclude our contributions and then propose some future work.

8.1 Contributions

The major contributions of this research work are:

- The reuse hypotheses and their metrics suites: Three hypotheses were proposed in this research regarding the relation between reusability and different quantitative attributes of object-oriented software components (inheritance, coupling, and complexity). The granularity of the hypotheses brought this research to a higher level than the previous work in its field. By verifying the hypotheses, we are able to either confirm what we have proposed or avoid going into the wrong directions in future work. The validations and evaluations of the predictive models displayed in chapter 7 appears that we have achieved the former.

- The building and selection of the best predictive models: The C4.5 machine learning algorithm was explored thoroughly during this research. In section 6.5 We also defined a method to build the best predictive models by experimenting different combinations of C4.5 options.
The method of validation and evaluation: In order to verify our hypotheses, we not only analyzed the predictive rule sets syntactically but also defined a set of quantitative measures to evaluate the correctness, completeness, and the accuracy of each predictive model. The data used for evaluation contains both the learning cases and the unseen test cases to make the validation more thorough and objective.

The process of finding useful reuse indicator: The most significant contribute of this research work is that we have established a methodology to explore the relationship between some specific OO software properties and the reuse measure. Such methodology begins with proposing a reuse hypothesis which is followed by metrics selection and extraction. A predictive model is then built and evaluated. The evaluation result of the classification model help us to verify the hypothesis proposed in the very beginning. The whole process can be reiterated to refine the hypothesis and its resulting reuse predictive model.

8.2 Future work

Due to the scope of this study and the time constraint, we are not able to reiterate the whole process so that we can refine the reuse hypotheses and thus to produce some even better predictive models. The interpretation of the rules needs further study. Some of the rules contradict with their corresponding hypotheses. We should find out why they are produced in the first place.

We would like to explore more about metrics extraction, which is the most time consuming part of this research. Without efficient metrics extraction this research can be paralyzed. Accurate and rapid metrics extraction not only facilitates the whole process of reuse detection, but also increases the predictability of the predictive models.

In a similar work in terms of methodology but nevertheless different in its needs and aims, Almeida et al. used another algorithm of learning, FOIL[25] with a more significant expressiveness[1]. In this algorithm, the example description language as well as the induced knowledge language (clauses) is of first order type. Thus, it has a better expressiveness than the algorithms based on the attribute-value language. It allows also to discover inter-metrics relationships. Moreover, FOIL gave better quantitative results than C4.5. This family of algorithms will be used in the continuation of our work.

Last but not least, we plan to use other OO systems with different sizes to improve the accuracy and to extend the scope of the obtained predictive models.
Bibliography


Appendix A

A validation and consulting tool: *Reusability Consult*

The purposes of developing the *Reusability Consult* (RC) are twofold. First, we need a tool that is able to apply the predictive rule models to the testing cases and produce accurate results. Secondly, such a process should be automated so that we can perform massive validation with data extracted from some large systems. RC takes the predictive rule sets and the testing cases as input and produces detailed report of the predicted reusability of each class in the testing set. If a case satisfies more than one rule, its predictive reusability is then ranked according to the corresponding confidence factors.

The structure and the functionality of the RC can be illustrated in Figure A.1. The task for the *lexical analyzer* is to parse the input files which contain the predictive rule models (section 7.1). After parsing, the whole *rule set* is divided into some *rules* that

![Diagram](image_url)
consist of a group of rule items. For instance, let us take the rule model displayed in Figure 7.1. The whole model is considered as a rule set. It contains 9 rules and a default rule. Each rule has its own rule items, a reuse class level, and a confidence factor. For example, rule 2 has two rule items which are: DIT > 1 and NOP > 6. Any case that fulfills with these two criteria is to be classified as reuse level 2 with confidence factor of 63.0%. The result of the lexical parser is then stored in the rule container. The rule container not only hold the information of parsed predictive rule models, but also provides service to the engine whose role is explained next. The most important method supported by the rule container is the function that checks if a given case is covered by the current rule. The header file presented below reflects the design of the rule container.

```
#include <stdio.h>

int main()
{
    // Main program
    return 0;
}
```

The RC engine accepts processed metrics values (as displayed in Figure 6.6) as input and then process each case with the help of the rule container. The engine checks each case with every rule in the rule set. If a rule covers the current case, the rule is pushed onto a temporary stack. After finishing processing one case, the temporary stack is sorted by the confidence factor of each rule in descending order. The result is also printed out on the screen, which can be stored into a file later. Let us present the original result of using RC verifying hypothesis 1 using testing data set. It contains predictive reuse level for 83 00 classes. Notice that the first part of the output file is the printout of the parsed rule model produced by the rule container. For each class, the output file provide its class number, class name, number of rules that this class is covered, its original reusability, and finally, the predicted reusability with corresponding confidence factor.
(Rule 1: HDG 2.00 HSP 4.00 RHP 6.00 class 2 [83.00])
(Rule 2: DIT 1.00 HDG 0.00 class 3 [38.00])
(Rule 3: DIT 0.00 HDG 2.00 DSN 16.00 HSP 4.00 RHP 6.00 class 3 [78.00])
(Rule 4: DIT 0.00 DSN 21.00 HSP 4.00 class 3 [62.00])
(Rule 5: DIT 0.00 HDG 0.00 DSN 11.00 HSP 1.00 class 3 [58.00])
(Rule 6: DIT 0.00 HDG 0.00 RHP 4.00 class 3 [71.00])
(Rule 7: DIT 2.00 class 4 [70.00])
(Rule 8: HDG 2.00 RHP 1.00 class 4 [63.00])
(Rule 9: HDG 10.00 class 4 [80.00])
Default class: 2

Result of Consult from data file:
C:\Project\Consult.txt data

----------class(1) [Action]. (2) rules found----------
claimed reusability [4]
Rule: [8] class: [4] confidence [83.00]
Rule: [8] class: [4] confidence [83.00]
----------Class(2) [ActionList]. (2) rules found----------
claimed reusability [4]
Rule: [8] class: [4] confidence [78.00]
Rule: [8] class: [4] confidence [83.00]
----------class(3) [Agent]. (1) rules found----------
claimed reusability [3]
Rule: [8] class: [4] confidence [60.00]
----------class(4) [Arguel]. (2) rules found----------
claimed reusability [4]
Rule: [8] class: [4] confidence [73.00]
Rule: [8] class: [4] confidence [83.00]
----------Class(5) [BelaifAction]. (1) rules found----------
claimed reusability [3]
----------class(6) [Belief]. (2) rules found----------
claimed reusability [6]
Rule: [8] class: [4] confidence [73.00]
Rule: [8] class: [4] confidence [83.00]
----------Class(7) [BeliefAction]. (1) rules found----------
claimed reusability [3]
----------class(8) [BeliefCond]. (1) rules found----------
claimed reusability [3]
----------Class(9) [BeliefMonitoring]. (2) rules found----------
claimed reusability [4]
Rule: [8] class: [4] confidence [73.00]
Rule: [8] class: [4] confidence [83.00]
----------Class(10) [BeliefMonitor]. (2) rules found----------
claimed reusability [4]
Rule: [8] class: [4] confidence [73.00]
Rule: [8] class: [4] confidence [83.00]
----------Class(11) [BinaryTimeExpression]. (1) rules found----------
claimed reusability [3]
----------Class(12) [Capability]. (2) rules found----------
claimed reusability [4]
Rule: [8] class: [4] confidence [73.00]
Rule: [8] class: [4] confidence [83.00]
----------Class(13) [Capability]. (2) rules found----------
claimed reusability [4]
Rule: [8] class: [4] confidence [71.00]
Rule: [8] class: [4] confidence [82.00]

74
Use default class [2]

Class[16] [CapabilityAction], no rules found matching claimed reusability [3]

Class[16] [CapabilityCond], [1] rules found matching claimed reusability [3]
Rule:[4] class:[3] confidence:[66.20]

Class[16] [CapabilityMonitoring], [2] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[17] [CommAction], [1] rules found matching claimed reusability [3]
Rule:[4] class:[3] confidence:[66.20]

Class[17] [CommCond], [1] rules found matching claimed reusability [3]
Rule:[4] class:[3] confidence:[66.20]

Class[19] [Commitment], [2] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[20] [CommitmentMonitoring], [2] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[21] [Confidential], [2] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[22] [DataPattern], [1] rules found matching claimed reusability [2]
Rule:[2] class:[2] confidence:[83.00]

Class[24] [EngineData], no rules found matching claimed reusability [4]
Use default class [2]

Class[26] [EVTCapabilityAction], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Rule:[6] class:[4] confidence:[73.00]

Class[37] [EVTCommAction], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[38] [EVTCommCond], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[39] [EVTExecution], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[43] [Execution], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[46] [ExecutionAction], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[47] [ExecutionCond], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[48] [ExecutionMonitoring], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]

Class[16] [EVTCapabilityAction], [1] rules found matching claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]
Class[33] [KreedEngineData], [1] rules found----------------------
claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[34] [KreedRule], [1] rules found----------------------
claimed reusability [2]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[35] [KreedRuleEX], [2] rules found----------------------
claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[36] [Artifact], [2] rules found----------------------
claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[37] [ArtifactEntry], [2] rules found----------------------
claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[38] [ArtifactModification], [2] rules found----------------------
claimed reusability [4]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[39] [FactPattern], no rules found matching----------------------
claimed reusability [2]
Use default class [2]
-------------------
Class[40] [FloatPattern], no rules found matching----------------------
claimed reusability [1]
Use default class [2]
-------------------
Class[41] [JsonObject], no rules found matching----------------------
claimed reusability [1]
Use default class [2]
-------------------
Class[42] [InferenceEngine], [1] rules found----------------------
claimed reusability [3]
-------------------
Class[43] [InvertBuffer], [2] rules found----------------------
claimed reusability [2]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[44] [Instantiation], [1] rules found----------------------
claimed reusability [2]
Rule:[6] class:[4] confidence:[73.00]
-------------------
Class[45] [IntegerPattern], no rules found matching----------------------
claimed reusability [1]
Use default class [2]
-------------------
Class[46] [Interval], no rules found matching----------------------
claimed reusability [1]
Use default class [2]
-------------------
Class[47] [InRule], no rules found matching----------------------
claimed reusability [4]
Use default class [2]
-------------------
Class[48] [LinkDensity], [1] rules found----------------------
claimed reusability [4]
Rule:[6] class:[4] confidence:[80.00]
-------------------
Class[49] [LooK], [2] rules found----------------------
claimed reusability [2]
Rule:[6] class:[6] confidence:[73.00]
-------------------
Use default class [2]

Class [51] (List), 2 rules found
claimed reusability [1]

Class [52] (Message), no rules found matching
claimed reusability [2]

Class [53] (Monitoring), 1 rule found
claimed reusability [3]

Class [54] (NegCondition), 1 rule found
claimed reusability [2]

Class [55] (NegEngine), 1 rule found
claimed reusability [4]

Class [56] (NegEngineData), 1 rule found
claimed reusability [4]

Class [57] (NegRule), 1 rule found
claimed reusability [2]

Class [58] (NegRuleEx), 2 rules found
claimed reusability [3]

Class [59] (OrdCondition), 1 rule found
claimed reusability [2]

Class [60] (OrdEngine), 1 rule found
claimed reusability [4]

Class [61] (OrdEngineData), 1 rule found
claimed reusability [2]

Class [62] (OrdRule), 1 rule found
claimed reusability [3]

Class [63] (OrdRuleEx), 2 rules found
claimed reusability [2]

Class [64] (Query), 2 rules found
claimed reusability [2]

Class [65] (Rule), no rules found matching
claimed reusability [2]

Use default class [2]

Class [66] (RuleAgent), 1 rule found
claimed reusability [4]
<table>
<thead>
<tr>
<th>Rule</th>
<th>Class</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule(1)</td>
<td>class:[3] confidence:[83.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(2)</td>
<td>class:[1] confidence:[73.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(3)</td>
<td>class:[5] confidence:[73.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(4)</td>
<td>class:[4] confidence:[73.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(5)</td>
<td>class:[6] confidence:[73.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(6)</td>
<td>class:[7] confidence:[73.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(7)</td>
<td>class:[8] confidence:[73.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(8)</td>
<td>class:[9] confidence:[83.00]</td>
<td></td>
</tr>
<tr>
<td>Rule(9)</td>
<td>class:[10] confidence:[83.00]</td>
<td></td>
</tr>
</tbody>
</table>