Interactive scalable condensation of reverse engineered UML class diagrams for software comprehension

Osman, M.H.B.

Interactive Scalable Condensation of Reverse Engineered UML Class Diagrams For Software Comprehension

Mohd Hafeez Osman

March 2015

Interactive Scalable Condensation of Reverse Engineered UML Class Diagrams For Software Comprehension

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 10 maart 2015
klokke 11:15 uur

door

Mohd Hafeez Osman
geboren te Perak, Maleisië
in 1979

Promotiecommissie

Promotores	:	Prof. dr. J.N. Kok	Universiteit Leiden
		Prof. dr. M.R.V. Chaudron	Chalmers and Gotëborgs Universitet
Copromotor	:	Dr. P. van der Putten	Universiteit Leiden
Commissieleden	:	Prof. dr. T.H. Bäck	Universiteit Leiden
		Prof. dr. Y.-G. Guéhéneuc	École Polytechnique de Montréal
		Prof. dr. J.J. Vinju	Technische Universiteit Eindhoven
		Dr. Ir. F. Verbeek	Universiteit Leiden

This research was financed by the government of Malaysia under the Federal Training Award (HLP).

Images used on the cover :

1. Victory Boogie-Woogie (1942-1944), by *Piet Mondrian*
2. Gray Tree (1911), by *Piet Mondrian*

Typeset by LaTeX, printed by Ipskamp Drukkers
ISBN: 978-94-6259-588-0

"to my family..."

Contents

Part I

Introduction and Background

Chapter 1

Introduction

In this chapter, we present the research context, the problem that we address and the goal of this research. This chapter also provides an overview of the research approach by summarizing the main research steps, the relations between these steps and their purpose. After reading this chapter, the readers should have a high-level understanding of the problem domain, our scoping, and our approach.

1.1 Research Context

Software design is a critical activity in software development. This design embodies the transition from a declarative requirement to a constructive representation that forms the basis for the implementation. Software design is essential to software implementation as well as to software maintenance. Documenting the software design could significantly help the later stages in the software development activity because the design is one of the critical documents to understand the software.

The effort and cost of software maintenance dominate the software development life cycle. The understanding of a software system is one of the most crucial tasks in software maintenance. More than 50% of the time consumed in software maintenance is used for software comprehension. Software documentation, including the software architecture or software design, is a highly useful material for system comprehension. Unfortunately, software documentation is often out-of-sync with the implementation [103],[172]. Reverse engineering is one of the options for recovering software architecture from the implementation code. This method suffers from several problems; one of them is that the resultant diagrams offer too detailed information. Recent Computer Aided Software Engineering (CASE) tools offer to leave out some types of information in software design diagrams (such as class diagrams) by leaving out attributes, operations and parameters. However, these tools are unable to identify the essential

information that helps the developer to understand or focus on specific concerns in system design. In addition, a controlled experiment by Fernandez-Saez et al. [59] found that many subjects did not consider reverse engineered diagrams to be helpful in maintaining software. From their study, they hypothesize that a large amount of data present in the reverse engineered class diagram overwhelms and demotivates users because it surpasses the human capacity for processing information (see e.g. [112]).

The research in this thesis focuses on the software comprehension activity in the software maintenance phase. We aim to provide a method and a tool for software developers to create an overview of their system. In addition, we aim to support the process of understanding software by enabling software developers to create multiple views of their system at various levels of abstraction that may differ for different tasks.

1.2 Problem Statement

When a new software developer is assigned to a maintenance task, several questions commonly arise as the software comprehension activity is started [93][92]. For instance, "Where to start?", "Which classes are important?", "How can I pick up the central classes needed for a more high-level, abstract view which is essential for understanding the model as a whole?" [142] and, "How to make this comprehension task easy?". Because the software documentation is often out-of-sync, these questions are difficult to answer; which makes the software comprehension task more challenging.

As mentioned in the previous section, reverse engineering techniques are capable of recovering a system's structure. A lot of CASE tools provide features that make the reverse engineering process easy to be performed by software developers. However, the resultant class diagrams constructed by these techniques typically contain such a large amount information that it obstructs design comprehension. Building a descriptive and understandable view of the software on the right level of abstraction is one of the most challenging tasks in reverse engineering [162]. This has led us to the following problem statements.

Problem Statement 1. *"How can reverse engineered class diagrams be simplified to assist software understanding?"*

We perceive a need for a framework to condense the reverse engineered class diagrams to improve its understandability. An automatic framework is desired to discover critical information, leaving out unnecessary information, and condense the reverse engineered class diagrams. However, to provide the aforementioned framework, the following issues also need to be addressed.

Problem Statement 2. *"What is the right level of abstraction of class diagrams?"*

We perceive a need for an interactive, scalable condensation of reverse engineered UML class diagram that provides the flexibility to developers to create multiple levels

of class diagram abstraction.

Hence, this research aims to address these issues by devising an automated framework by simplifying UML class diagrams to assist software comprehension. The following subsection explains our research objective to address these issues.

1.3 Research Objective

The central objective of this research is to devise an automated framework for simplifying UML class diagrams to assist the software comprehension task. We use reverse engineered class diagrams (obtained by static analysis) as the primary source of information about a system.

To achieve this, our research focuses on discovering a suitable method to identify the critical and non-critical information in reverse engineered class diagrams. We also aim to provide a prototype implementation of this method through a tool. This prototype should demonstrate the feasibility of the approach. The tool should be interactive and the condensation of class diagrams should be scalable to allow the software developer to generate views of designs at their desired levels of abstraction.

To accomplish our objectives, the following research questions (**RQ**) have been formulated:

Main RQ: *What method of condensing of reverse engineered class diagrams helps developers to understand the design of software systems?*

To answer the **Main RQ**, we need to answer the following RQs:

- **RQ1:** *Which information in class diagrams do developers find important for understanding software designs?*
- **RQ2:** *Which object-oriented design metrics do developers find most indicative for class importance?*
- **RQ3:** *How to automatically condense class diagrams using object-oriented design metrics?*
- **RQ4:** *Can the automatic condensation of class diagrams be enhanced by using class names?*
- **RQ5:** *Does our automated framework for condensing class diagrams help developers to understand the design of software systems?*

1.4 Research Methods

The main objective of this research is to discover a method of enhancing the comprehension of reverse engineered class diagrams. To accomplish this goal, we apply various research methods, including: surveys [137], case studies [46] and experiments [148]. Details about the research methods used are provided in Table 1.1.

Table 1.1: *Research Methods used in this Research*

Chapter	Methodology	Primary Objective	Primary Data
3	Field Study	Descriptive	Qualitative
4	Experiment	Descriptive	Quantitative
5	Survey	Descriptive	Qualitative
6 & 10	Survey	Descriptive	Quantitative
7 & 8	Experiment	Validation	Qualitative

In summary, we used surveys for eliciting information on how the software engineers think classes diagrams could be simplified. An experiment is used to explore the state-of-the-art of reverse engineering class diagrams. A field study [185] is used to explore the usage of UML diagrams in open source software development. We used experiments to explore and validate the effectiveness of some class condensation techniques that we developed. We also used the survey method to validate our proposed automated framework for condensing class diagrams.

We provide our experiments' material (online) for the purpose of external replication and future research ([6] [122] [134]).

1.5 Roadmap

This section presents an outline (see Figure 1.1) of the chapters in this thesis. We summarize the purposes of each chapter and relate the chapters to the research questions. Also, we relate the chapters to our publications.

- **Chapter 2: Definition.** The purpose of this chapter is to define the principal concepts used in this research. We briefly describe the Unified Modeling Language (UML) and class diagrams. Also, we describe the concepts of forward and reverse engineering, the basics of machine learning and explain the notion of software comprehension.

- **Chapter 3: UML Usage in Open Source Software Development.** The purpose of this chapter is two-fold: i) To present the examples on the use of UML diagrams in Open Source Software Development (OSSD) and ii) To find suitable case studies for automatic condensation of class diagrams research. For this purpose, we select ten OSSD projects from different types of domains. We assess the UML usage of OSSD projects, the level of detail (LoD) and the frequency of updating diagrams. Our findings also cover the application of UML modeling in different level of detail for different purposes, a change in focus on types of diagram used over time, and findings on how the size of models relates to the size of the implementation.

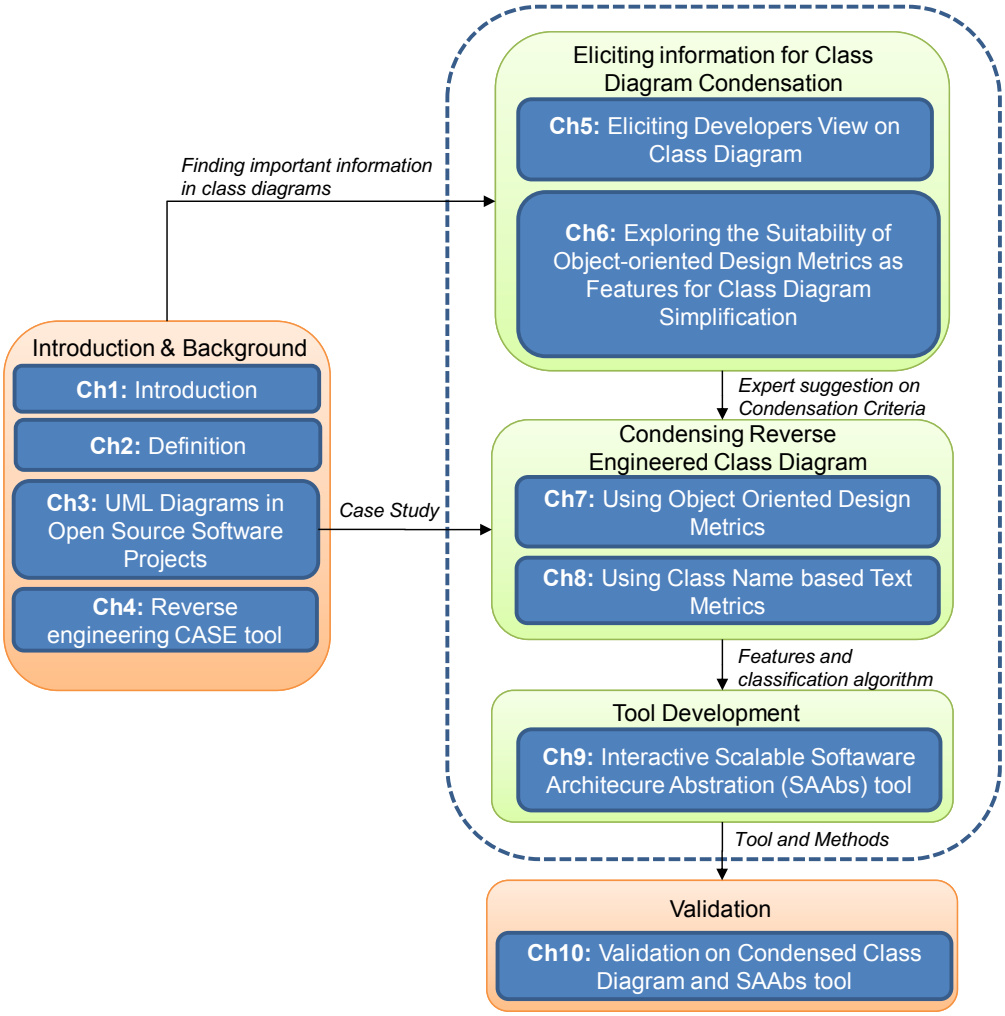


Figure 1.1: *Thesis Roadmap*

This chapter is a more detailed version of the following publication:

- Hafeez Osman and Michel R.V. Chaudron (2013). **UML Usage in Open Source Software Development : A Field Study.** *In Proceedings of the 3rd International Workshop on Experience and Empirical Studies in Software Modelling (EESSMod 2013), pages 23-32, Miami, USA*

- **Chapter 4: Assessing the Correctness and Completeness of UML CASE tools in Reverse Engineering.** The main purpose of this chapter is to demonstrate the state-of-the-art of CASE tools for reverse engineering of source code into

class diagrams. We assess the strengths and the weaknesses of the reverse engineered class diagrams constructed by eight common CASE tools. We compare and evaluate the types of input, the types of reverse engineered diagrams that could be constructed, and the quality of resulting diagrams. This chapter covers information about the correctness, completeness and the quality of the reverse engineered class diagrams (as constructed by CASE tools). The results provide a baseline of current reverse engineering of class diagrams by CASE tools.

This chapter is adapted from the following publications:

- Hafeez Osman and Michel R.V. Chaudron (2011). **An Assessment of Reverse Engineering Capabilities of UML Case Tools.** *In Proceedings of the 2nd Annual International Conference on Software Engineering and Applications (SEA 2011), pages 7-12, Singapore*
- Hafeez Osman and Michel R.V. Chaudron (2012). **Correctness and Completeness of CASE tools in Reverse Engineering Source Code into UML Model.** *GSTF Journal on Computing vol.2, num.1, pages 193-201*

- **Chapter 5: Eliciting Developer's Views on Simplifying Class Diagrams.** In this chapter, we aim to discover how to simplify class diagrams in such way that the system is easier to understand. For this purpose, we conduct a semi-structured survey to gain knowledge about the criteria that developers believe are relevant for including or excluding in class diagrams. The results of this survey suggest what are the important elements in a class diagram.

  This chapter answers **RQ1** and it is a more detailed version of the following publication:

  - Hafeez Osman, Arjan van Zadelhoff, Dave R. Stikkolorum and Michel R.V. Chaudron (2012). **UML Class Diagram Simplification: What is in the Developer's Mind?** *In Proceedings of the 2nd International Workshop on Experience and Empirical Studies in Software Modelling (EESSMod 2012), pages 31-36, Innsbruck, Austria*

- **Chapter 6: Exploring the Suitability of Object-oriented Design Metrics as Features for Class Diagram Simplification.** The purpose of this chapter is to identify suitable design metrics that influence the determination of class inclusion and exclusion. We conduct a survey to investigate the suitability of object-oriented design metrics (from software documents) in deciding on the inclusion and exclusion of classes from class diagrams. The results indicate what software design metrics are most important to users to decide whether to include a class in a class diagram.

  This chapter answers **RQ2**. It is a more detailed version of the following publication:

  - Hafeez Osman, Arjan van Zadelhoff and Michel R.V. Chaudron (2012). **UML Class Diagram Simplification - A Survey for Improving Reverse Engineered Class Diagram Comprehension.** *In Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2013), pages 291-296, Barcelona, Spain*

- **Chapter 7: Condensing Reverse Engineering Class Diagram using Object-Oriented Design Metrics.** The purpose of the study in this chapter is to investigate the usefulness of object-oriented design metrics as features for identifying class inclusion and exclusion. To this end, we conduct an experiment for condensing reverse engineered class diagrams on the basis of software design metrics. We use object-oriented design metrics as features for applying machine learning classification approaches to classifying the classes for inclusion and exclusion. The machine learning is used because it provides an automated method for the classification process. Nine OSSD projects are used as case studies. This experiment also identifies the classification algorithms that perform best for this purpose.

  This chapter answers **RQ3** and it is adapted from the following publication:

  - Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2013). **An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams.** *In Proceedings of the 29th International Conference on Software Maintenance (ICSM 2013), Eindhoven, the Netherlands*

- **Chapter 8: Condensing Reverse Engineered Class Diagrams through Class Name Based Abstraction.** The purpose of this chapter is to improve the classification of class inclusion and exclusion by using class names. We formulate text metrics based on the frequency of occurrence of words in class names. We explore multiple combinations of features and compare the results with the previous outcomes (Chapter 7). The evaluation is performed using 10 OSSD projects. This chapter presents the improvement of class inclusion and exclusion classification, by using text and object-oriented design metrics as features.

  This chapter answers **RQ4** and it is adapted from the following publication:

  - Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2014). **Condensing Reverse Engineered Class Diagrams through Class Name Based Abstraction.** *In Proceedings of the 2014 World Congress on Information and Communication Technologies (WICT), Malacca, Malaysia*

- **Chapter 9: Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams.** In this chapter, we demonstrate our automated Software Architecture Abstraction (SAAbs) framework for simplifying class diagrams based on class inclusion/exclusion (Chapter 7 and 8). The SAAbs framework applies a machine

learning classification algorithm to produce a class importance ranking for all classes in a reverse engineered class diagram. This ranking is used for scalable abstraction and visualization of the class structure of the system. We created a tool that allows developers to interactively explore a reverse engineered class diagram at multiple levels of abstraction.

Part of this chapter is adapted from the following publication:

- Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2014). **Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams.** *In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014), Jeju, Korea*

- **Chapter 10: Validation.** In this chapter, we conduct a user study to validate the SAAbs framework and tool in providing a platform for assisting developers to comprehend reverse engineered class diagrams. This chapter aims at i) discovering the understandability of condensed class diagrams, ii) finding whether the condensed class diagram generated by this approach is helpful in understanding the software design and, iii) eliciting the usefulness of the SAAbs tool in assisting software developers to understand the software.

  This chapter answers **RQ5**. Part of this chapter is adapted from the following publication:

  - Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2014). **Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams.** *In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014), Jeju, Korea*

- **Chapter 11: Conclusion.** In this chapter, we summarize the results, draw conclusions and discuss future work.

Chapter 2

Definitions

In this chapter, we define common terms used in this research. We start with describing the term 'software comprehension' because it is a common theme throughout this thesis. We also describe 'cognitive design elements for software exploration' because they are the basis for representing the results of our approach in a graphical visualization. Also, 'forward-' and 'reverse engineering' process are explained because we use the information in these processes as the primary input in our studies. We briefly describe UML because this design notation is the central notation used in this research. We explain the basics of machine learning because this method is used to classify the importance of elements of the software design.

2.1 Software Comprehension

Software comprehension (also known as program-/code-/system- comprehension) is a critical activity in software maintenance. Software comprehension is a complicated activity which requires a lot of time, effort and cost. In particular, studies have shown that more than 50% of the time spent in software maintenance is spent on trying to understand source code [20][23]. Software developers must familiarize themselves when they are new to a system, but it is also the case that they must familiarize themselves with new parts of a system as the system evolves. Table 2.1 lists several definitions on program comprehension as found in the literature. Based on these definitions, we define program comprehension in this research as:

Definition 2.1. "Program comprehension is a process of understanding a program based on available software artifacts (such as documentations, as well as source code)."

Biggerstaff (1993) [25] characterizes what aspects of the software must be understood through the comprehension process: *"A person understands a program when they*

Table 2.1: *Definitions of Program Comprehension*

Author(s)	Definitions
Rugaber (1995) [88]	*"Program comprehension as the process of acquiring knowledge about a computer program"*
Cimitile (2000) [89]	*"Program comprehension is the process of acquiring sufficient knowledge about a software artifact so as to be able to successfully accomplish a given task"*
Ng et al. (2004) [116]	*"Program comprehension is the process of understanding a program through feature and documentation analysis"*
Maalej et al. (2014) [107]	*"Program comprehension is the activity of understanding how a software system or a part of it works"*

are able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."

Program comprehension is a tedious challenge that requires a lot of effort. Hence, we believe that facilitating program comprehension will help software developers in this activity. Several models have been proposed that characterize the cognitive process of program comprehension. We discuss these program comprehension models in the next subsection.

2.1.1 Program Comprehension Model

A software developer's approach to program comprehension may be bottom-up, top-down or a combination of both. This is relevant in the context of this thesis as it led us to develop methods to view class diagrams at different levels of abstraction. We briefly discuss these program comprehension models below.

Bottom-up Comprehension

Bottom-up comprehension models denote the bottom-up building of system understanding: by reading the code and then rationally chunking or grouping these statements or representation into higher-level abstractions. These chunks (or abstractions) are aggregated further until the entire system is comprehended [152].

The Pennington comprehension model [135] describes two program abstractions formed by software developers during comprehension [43]: (1) A *program model* is a low-level abstraction consisting of knowledge of operations at a level close to the surface of the program code and of control flow relations indicating the flow of execution, and (2) A *domain model* (or *situation model*) is a higher-level abstraction comprising knowledge of data flow and functional relationships. This comprehension model is frequently

chosen when the source code or domain of the system is not familiar to the software developer.

Top-down comprehension

The top-down comprehension model (based on Brooks's model [37]) denotes a hypothesis-driven comprehension approach. This approach begins with the software developers making a general hypothesis about the program's function. Such hypothesis are formed based on information outside the program code such as design documents or system descriptions. This initial hypothesis leads software developers to anticipate to certain structures (objects and operations) in the program, producing another level of more distinct hypotheses (sub-hypotheses). At this point, the software developer has concrete things to look for in the program code, for which hypothesis verification is attempted [43].

In contrast to the bottom-up approach, the top-down comprehension is usually used for a system and a domain that is familiar to the software developer.

Integrated Model

Von Mayrhauser (1993) [173] introduced an integrated model that combines the top-down and bottom-up model. This model is motivated by the argument that software developers may switch flexibly between top-down to bottom-up comprehension models depending on the situation. In addition to the top-down and bottom-up comprehension model, the integrated model also involves the knowledge base (typically known as human mind) that stores (1) any new information acquired straightforwardly from the system; or (2) indirect information. Figure 2.1 illustrates this model and more detail on the terms and terminology of this figure can be found in [174].

2.1.2 Cognitive Design Elements for Software Exploration

The time consumed for understanding existing implementation code is a significant proportion of the time needed for maintaining, debugging and reusing the existing code. This explains the importance of tool support for software comprehension. Tools in this area can be characterized as being a software visualization tool or a software exploration tool. Schäfer et al. [150] indicated that software visualization tools (e.g., [100],[86],[49]) assemble visualization techniques to intensify understanding, while software exploration tools (e.g. [57],[145]) offer an essential software navigation facility (searching/browsing/summarizing/condensing). However, there is no concrete boundary between these categories. In this research, we focus on software exploration tools as we aim to provide an automated tool to explore the software design based on the reverse engineered class diagram.

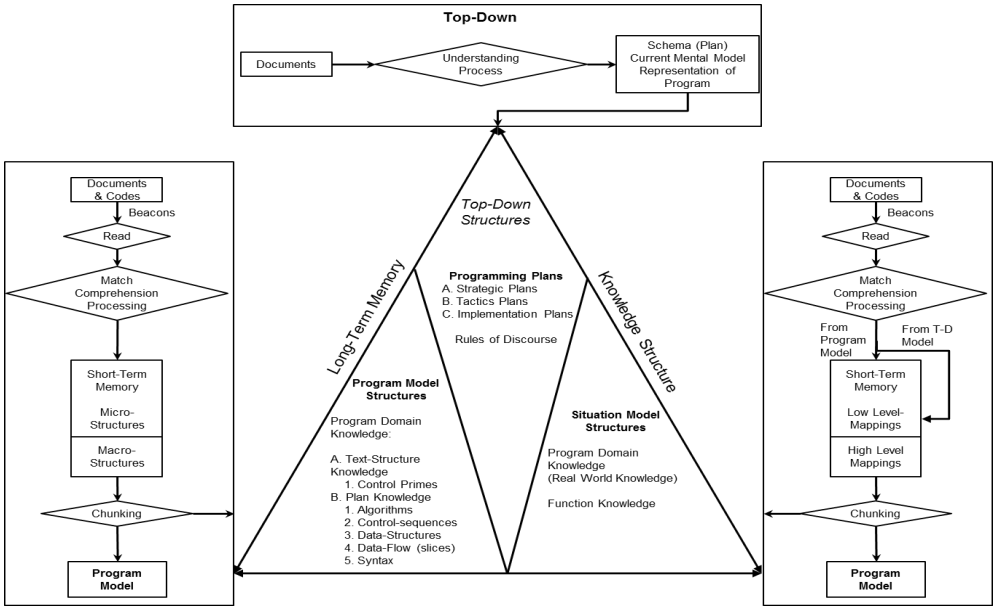


Figure 2.1: *Integrated Model [174]*

A common feature in software exploration tools is the graphical representation of the system structure together with the corresponding source code. Storey et al.[157] introduced a set of guidelines for software exploration tools (as shown in Figure 2.2). These guidelines are called '*cognitive design elements*'. From the guidelines that she suggests, we prioritize a number which we aim to fulfill in our approach. The tool developed in this research focuses on comprehension of software design. Hence, we would like to incorporate the following guidelines to our tool: **E3**: Provide an abstraction mechanism; **E4**: Support goal-directed hypothesis-driven comprehension; **E5**: Provide an adequate overview of the system architecture at various levels of abstraction; **E6**: Support the construction of multiple *mental models*[1] (domain, situation, program); **E11**: Indicate the maintainer's current focus; and **E15**: Provide effective presentation style.

2.2 Forward and Reverse Engineering

In software engineering, the software development life cycle (SDLC) is a schema that characterizes the process of developing software. In other words, it is a conceptual model used in software project management that describes the stages involved in system development from preliminary feasibility review through maintenance. The

[1] A *mental model* describes developer's mental representation of the system-to-analyze

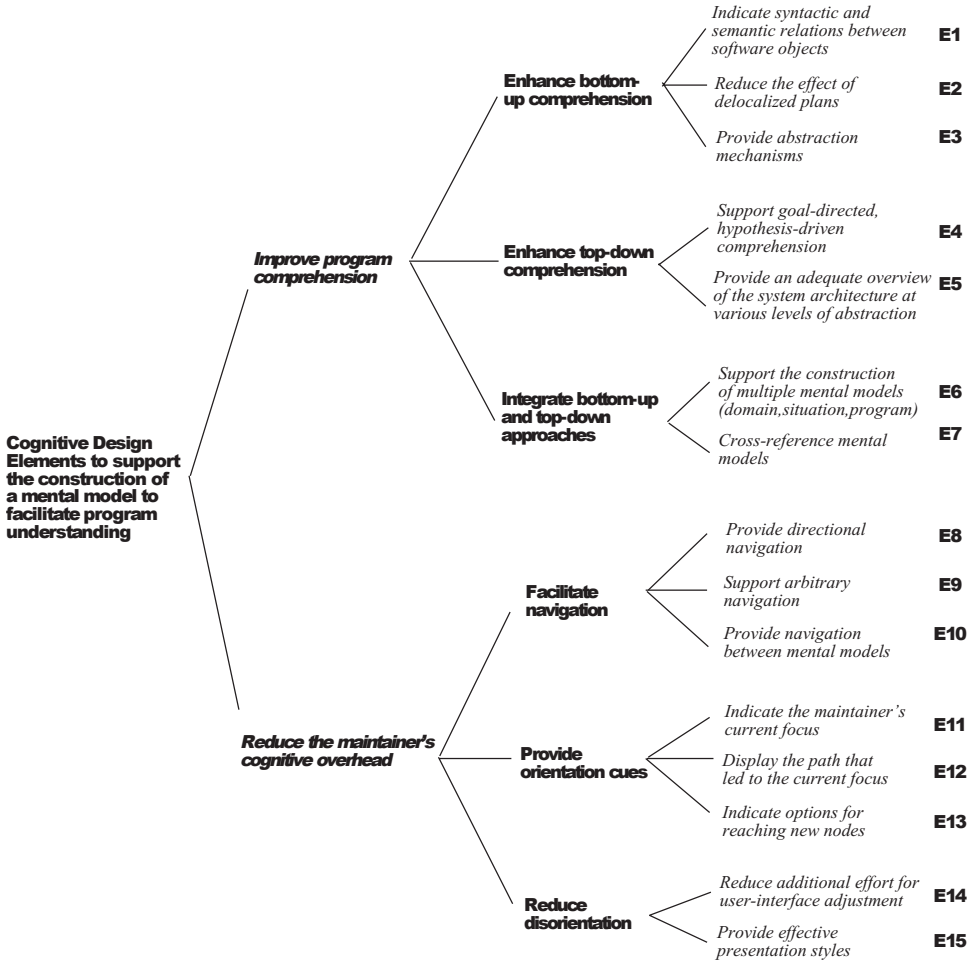


Figure 2.2: *Cognitive Design Elements for Software Exploration [156]*

SDLC describes software development stages (at a high level) as: analysis, design, implementation and maintenance. In this research, we focus on three stages of the SDLC as illustrated in Figure 2.3 (stages that are coloured with blue). We use artifacts from the design and implementation stage (as our input) to produce a result (output) for the usage in the maintenance stage. The term software maintenance (from SDLC) is typically used in this research. The following definition of software maintenance by IEEE std 1219-1998 [1] is used in this research:

Definition 2.2. "Software maintenance is a modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

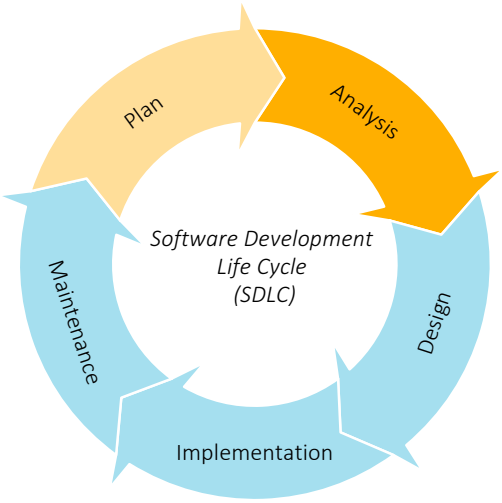


Figure 2.3: *The Software Development Life Cycle*

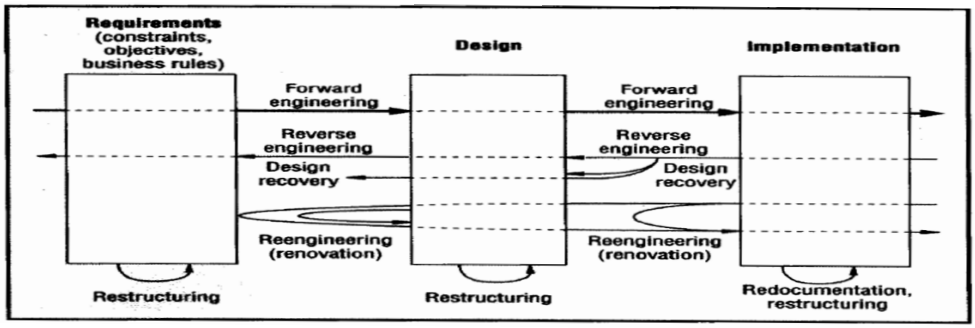


Figure 2.4: *Relationship between Forward Eng., Reverse Eng. and Other Related Terms [41]*

Regarding these stages (see Figure 2.4), there are two processes related to this research: Forward Engineering and Reverse Engineering. The following subsections briefly explain these processes.

2.2.1 Forward Engineering

The following definition of forward engineering by Chikofsky and Cross [41] is used in this research.

Definition 2.3. *"Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."*

The term forward design (FD) is used to indicate a design produced in a forward engineering routine (constructed after the requirement stage). The FD usually addresses the functional requirement of the system (mostly specific to the system domain) and also the non-functional requirements.

2.2.2 Reverse Engineering

The following definition of reverse engineering by Chikofsky and Cross (1990) [41] is used in this research.

Definition 2.4. *"Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level abstraction."*

In general context, reverse engineering is the process of recovering knowledge of software, based on the existing software artifacts. There are different types of techniques can be used in the reverse engineering process, most importantly: static- and dynamic analysis. We describe these next.

2.2.3 Static and Dynamic Analysis

This subsection briefly describes the definition of static and dynamic analysis that we use in this research. For static analysis, the following definition by Jarzabek (2007) [85] is used in this research.

Definition 2.5. *"Static analysis means an analysis of a program text, without executing a program, such as is typically done by a compiler front end."*

In most cases, the analysis is performed on the source code, and in other cases, it is performed on the object code and execution file. Static analysis is suitable for recovering the system structure (e.g. structural design).

For dynamic analysis, the following definition by Bell (1999) [19] is used in our research:

Definition 2.6. *"Dynamic analysis is the analysis of the properties of a running program. Dynamic analysis derives properties that hold for one or more executions by examination of the running program."*

An advantage of using dynamic analysis is its ability to detect objects dependencies (at runtime). However, dynamic analysis cannot guarantee to discover the complete functionality of the system for an overview for the developer. Amongst others, this is because dynamic analysis can only pass through a limited subset of functions in the systems within any practical time-bound. In addition, the source code (and hence modules) that are passed through typically depend on the set of input values, which for most practical purposes has infinitely many combinations. In this thesis, we work

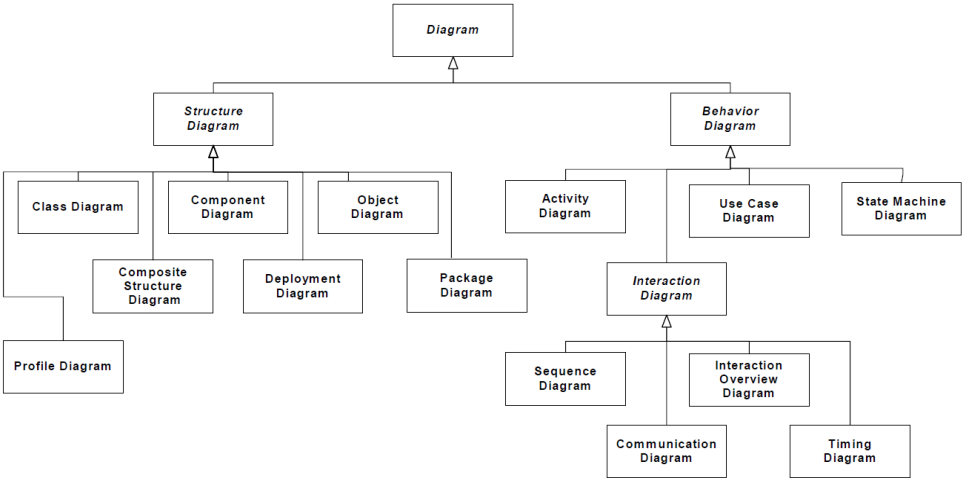


Figure 2.5: *Taxonomy of UML Version 2.4 [68]*

purely with static analysis for the recovery of the structure of software designs from source codes.

Next, we explain the UML notation which is used for representing this static design structure of software.

2.3 The Unified Modeling Language

The Unified Modeling Language (UML) is a graphical notation intended to provide a standardized communication tool in software development. In practice, software is almost always developed by a group of software engineers. For this reason, a communication tool is critical to ensure good communication because the software design should be well understood by everyone working on the project.

UML was introduced in 1997, and was developed based on existing object-oriented design methods, namely the object-modeling technique (OMT) [146], Booch [31], and Object-Oriented Software Engineering (OOSE) [83]. There are two kinds of UML Diagrams: structure- and behavior diagrams. A taxonomy of UML diagrams is shown in Figure 2.5. The structure diagrams demonstrate the structural parts of the system at diverse levels of abstraction as well as how structural parts are related to one another. On the other hand, the behavior diagrams demonstrate the behavior of a system, which could be portrayed as an arrangement of actions of the system over time. Through supporting both these structure and behavior diagrams, UML provides a graphical representation of a system during design, implementation, as well as the maintenance phase. The next subsection describes the UML-related terms that are commonly used in this research.

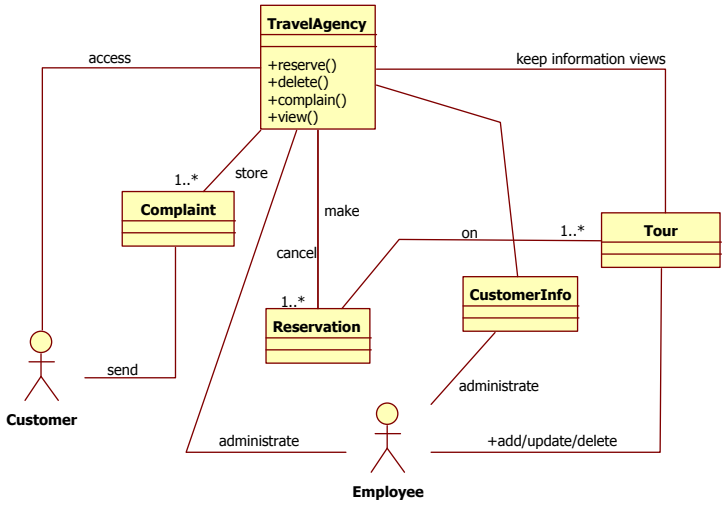


Figure 2.6: *Tours Online Class Diagram (Domain Analysis)*

2.3.1 UML Class Diagram

The most common diagram to demonstrate the structural view of a system is the class diagram [147]. This view illustrates a collection of classes, possibly interfaces, and relations between classes. Relationships that hold between classes convey critical information about the design. The basic types of relationships in UML are association (including aggregation and composition) and generalization (also known as inheritance). These relations provide the foundation of the system structure.

Class diagrams can be used throughout the software development life cycle (SDLC), due to the fact that this diagram may carry diverse types of information - depending on the SDLC processes and on the level of detail being considered. At the beginning of the SDLC, the class diagram may be used to reflect the software requirements (*domain analysis class diagram*). As development progresses, class diagrams can be used to represent information that is more relevant to the construction of the system (*design level class diagram*[2]). During or after the implementation of source code, a class diagram may be recovered using reverse engineering techniques. Such a reverse engineered class diagram is closely based on the source code and reflect the fine-grain implementation structure of software systems. We call such reverse engineered class diagrams as RE-CD. Figure 2.6, 2.7 and 2.8 illustrate these different types of class diagrams. These examples of class diagrams are taken from Jalloul [84].

[2]Depending on the level of detail (LoD) of the class diagram, *design level class diagrams* may be turned to *code level class diagrams* if a high LoD is applied.

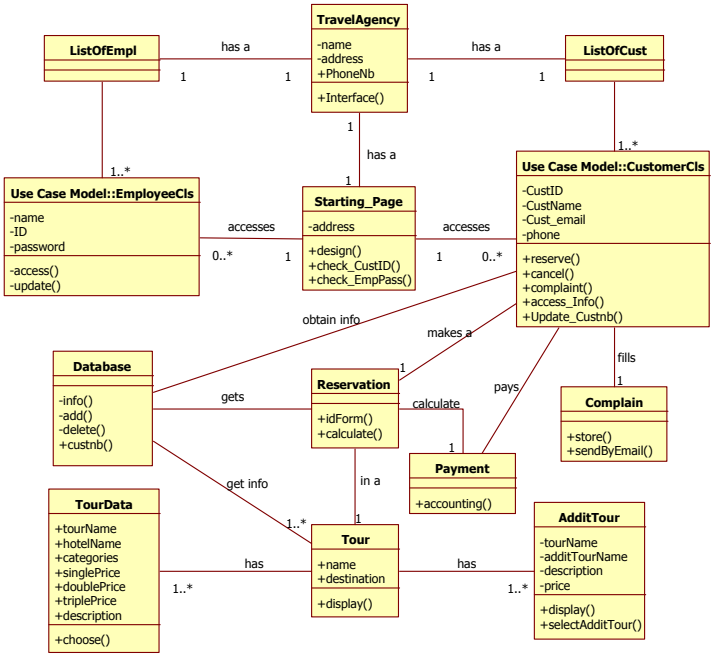


Figure 2.7: *Tours Online Class Diagram (Design Level)*

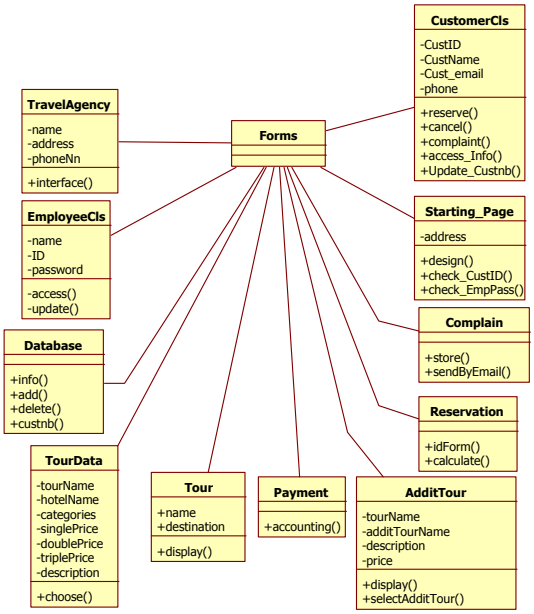


Figure 2.8: *Tours Online RE-CD (Code Level)*

2.3.2 UML Class Diagram for Software Comprehension

A graphical representation (e.g. UML class diagram) can help software engineers to comprehend large-scale systems. However, their effectiveness is subject to the syntax and semantics of UML, spatial diagram layout and domain knowledge [166]. Yusuf et al. [183] show that experts (in class diagrams) tend to use such things as stereotype[3] information, colouring and layout to facilitate more efficient exploration and navigations of class diagrams.

In this research, we focus on RE-CDs that are close to source code (*code level class diagrams*). The stereotype information is not available in RE-CD and hence this information is not used in our approach.

The work by [160], [183], [71], [151], [158] and [159] demonstrate the effect of layout on system comprehension. However, we believe that the choice of layout is subjective and highly depends on the user expertise and purpose of using the diagram. Thus, we did not cover the layout of the RE-CD in this thesis. The aspect of layout remains open or for future research. Nonetheless, in our research, we apply a colouring technique to highlight those classes that are important in the class diagram. This colouring technique aims to help the software developers to focus on the important classes.

2.3.3 XML Metadata Interchange

XMI stands for Extensible Markup Language (XML [176]) Metadata Interchange. It is a standard for representing UML models using XML. The current version released by the OMG is XMI 2.4.1 which has been formally published by the International Organization for Standardization (ISO) as ISO/IEC 19509:2014 Information technology – Object Management Group XML Metadata Interchange (XMI)[69].

The objective of XMI is to allow simple interchange of metadata between UML modeling tools and Meta Object Facility (MOF)-based repositories within distributed heterogeneous environments. The standards that are related to XMI are the following:

1. **UML** – Unified Modeling Language (ISO/IEC 19505)
2. **MOF** – Meta Object Facility (ISO/IEC 19508)
3. **XML** – eXtensible Markup Language [176]

The MOF defines a standard metamodel for applications, allowing UML models to be interchanged among tools and repositories; XMI standardizes the format for these interchanges [66]. It utilizes XML schemas to describe object-oriented models and enable interoperability between UML-based tools. XMI is flexible. Thus, the XML representation can be tailored to suit the user requirements. Most of the UML tools extend the XMI format with their proprietary information, which result in that other UML-based tools can not completely and correctly read the XMI file. This issue has

[3] A stereotype is a special type of class that represents a domain-specific concept. Graphically, a stereo-type class can be adorned with a special graphical form or decorations so that it stands out from generic classes.

been raised by Stevens [155] in 2003, but it still exists after more than ten years. In this research, we aim to parse as much XMI flavours and versions as possible; to provide an automated tool that usable for many XMI formats and hence UML tools. Finding a way to solve this issue is one of the practical challenges in this research.

2.4 Machine Learning

This research aims at building a method to decide: what classes could be included and what classes could be excluded in class diagrams in order to facilitate system comprehension. Two approaches can be used to build this method:

1. *Rule-based Approach*
   A rule-based system consists of if-then rules, facts, and an interpreter controlling the application of the rules. Conventional rule-based expert systems, use human expert knowledge to solve real-world problems that normally would require human intelligence [12] and,
2. *Machine Learning Approach*
   The machine learning approach is useful for domains where humans might not have the knowledge needed to develop effective algorithms, where the program must dynamically adapt to changing conditions [113].

Based on the datasets (see Chapter 3), we discovered that the machine learning approach is more suitable to be used for our research. The reason is that there is no explicit knowledge available in the dataset. Furthermore, the dataset is coming from different types of domain and sizes. It is difficult for such context-sensitive problem to be solved by using a rule-based approach. Also, the following reasons motivate us to choose machine learning as the approach for class inclusion/exclusion selection:

- It provides algorithms that may facilitate to automatically[4] classify the important classes in a class diagram based on the training data (in our case, training data are gathered from forward design and RE-CD);
- Human resources are not required to formulate rules. Therefore, it may avoid the inefficiency of human learning [16].
- It considers context and it can utilize multiple sources of knowledge to formulate the classification rules.
- It can adapt if new information becomes available.

Next, we explain the definitions of machine learning, types of machine learning, machine learning classification algorithms and performance measure for classification algorithms.

[4] With no or little human intervention.

2.4.1 Definition of Machine Learning

The common definitions of machine learning are the following:
Arthur Samuel (1959) [149][113] defines machine learning as:

Definition. *" A field of study that gives computers the ability to learn without being explicitly programmed."*

This definition outlines the basic concept of machine learning. Later, Mitchel (1997) [113] introduces a further formalized definition of machine learning as shown in the following.

Definition. *"Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*

However, Witten et al. (2005) [178] more focus on the 'descriptions' model (from examples). They define machine learning as:

Definition. *"The acquisition of structural descriptions from examples. The kind of descriptions found can be used for prediction, explanation, and understanding."*

There are several types of machine learning provided to solve problems (depends on the purpose and available data). The types of machine learning are described in the next section.

2.4.2 Types of Machine Learning

The following describes four major types of machine learning algorithms:

- *Supervised machine learning* is the search for algorithms (see Figure 2.9) that reason from externally supplied instances to produce general hypotheses, which then make predictions about future instances [98]. The training data (input) have a known label or predefined result, for example, a binary label of buy/not buy in the stock market. Through the training process, a model is constructed to make predictions of test instances (data). Examples of supervised learning tasks are classification and regression.
- *Unsupervised machine learning* learns to characterize certain input pattern in a fashion that reflects the statistical structure of the overall collection of input patterns [45]. The input data are unlabeled, and no predefined results are provided. The goal of unsupervised learning is to find some kind of structure in the data. An example of a common unsupervised learning task is clustering.
- *Semi-supervised learning* is halfway between unsupervised and supervised learning [38]. The input data is a mixture of unlabeled and labeled sample. The purpose of semi-supervised learning is to discover how combining labeled with unlabeled data may change the learning behavior, and design algorithms that benefit from such a combination.

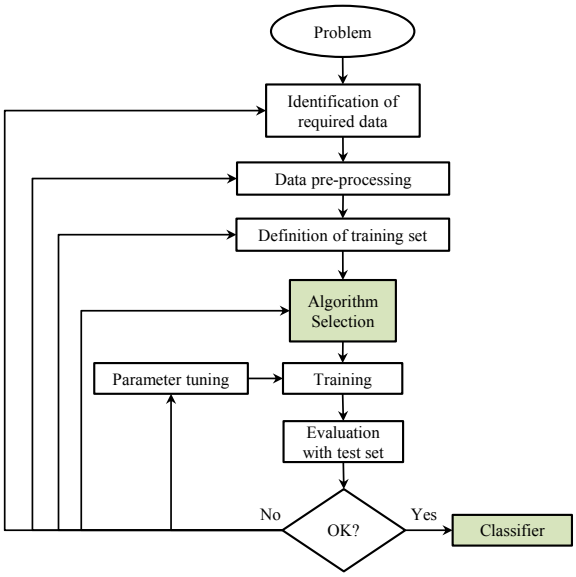


Figure 2.9: *The Process of Supervised Machine Learning [98]*

- *Reinforcement machine learning* is the learning of a mapping from situations to actions so as to maximize a scalar reward or reinforcement signal [161]. The learner is not told which action to take, but must discover which action results in the best reward by trying them. The action may affect not only the immediate reward, but also the next situation through all subsequent rewards.

This research applies supervised machine learning classification algorithms to decide what classes could be included and what classes could be excluded (omitted) in class diagrams. Next, we explain the supervised machine learning classification algorithms that are used in our experiments.

2.4.3 Machine Learning Classification Algorithms

This subsection focuses on several supervised machine learning classification algorithms that we believe are suitable for the purpose of this research. As illustrated in Figure 2.9, a selection of classification algorithm(s) is needed to make sure our proposed framework uses the algorithm(s) that fit with the datasets and the purpose of research. Prior to making a selection of the classification algorithms, several exploratory experiments on a wider range of algorithms need to be conducted. In this research, we do not expect that there will be a single silver bullet algorithm that will outperform all others across all sets of problems. Also, we are not just interested in a single algorithm that scores a top result on a given problem, but are looking for sets of

classifiers[5] (i.e. classification algorithms) that produce robust results across domains. In this way, algorithms become more portable across problems with very different rates of inclusion of classes in designs. We also aimed for a mix of classifiers in terms of expected bias (what relationships can be captured) and variance (does the prediction change when trained on different random samples) [171].

As discussed, we want to use a diverse set of algorithms representative for different approaches. For example, Decision Trees, Stumps, Tables and Random Trees or Forests all divide the input space up in disjoint smaller sub-spaces and make a prediction based on the occurrence of positive classes in those sub-spaces. K-Nearest-Neighbour (k-NN) and Radial Basis Functions (RBF) Networks are similar local approaches, but the sub-spaces here are overlapping. In contrast, Logistic Regression and Naive Bayes model parameters are estimated based on potentially large numbers of instances and can thus be seen as more global models. The nine classification algorithms we consider are described in Table 2.2 (refer [178] for more explanation).

Most of the classification algorithms in our experiments are designed to produce a predicted outcome class label[6] for each test instance [58]. However, this research aims to produce ranking of classes on the importance; hence, we are more interested in the *classifier score* for every instance rather than just a set of instance classification labels. A high *classifier score* of a class indicates the class is important while a lower *classifier score* indicates the class is less important.

The classification algorithm(s) for this research are selected based on the classification performance and their robustness to all datasets. We explain the performance measure of classification algorithms in the next section.

2.4.4 Performance Measure For Classification Algorithms

A performance measure of machine learning classification algorithms can be derived from a confusion matrix (as shown in Table 2.3). Several performance measures to compare classification algorithms (formulated based on the confusion matrix) are described in Table 2.4 (refer [178] for more detail). Our datasets used in this research are typically imbalanced (i.e. low proportion of classes in forward design and high proposition of classes in RE-CD, as shown in Chapter 3, 7 and 8). Hence, the common performance measures listed in Table 2.4 do not fit for our purpose. Referring to the confusion matrix example in Table 2.3, the overall success rate (accuracy) is 95.24%. It seems that the algorithm performs an excellent prediction. The 95.24% is calculated by taking the sum of correct prediction divided by the overall number of predictions. The percentage of correct prediction for TN is 98.8%, while TPR is 25%. The resulting prediction performance for TP is very low, even though overall correct prediction is very high.

[5] In this thesis, *classifier* refers to classification algorithms models, not the term classifier in UML.

[6] A binary-classification that attempts to produce 'Yes' or 'No' class labels.

Table 2.2: *The nine classification algorithms*

Algorithms	Description
Decision Table	A Decision Table consists of rows and columns that associate a set of conditions or tests with a set of actions. The machine learning tool used in this research - Waikato Environment for Knowledge Analysis (WEKA) [76] uses a simple Decision Table Majority (DTM) classifier.
Decision Stumps	Decision Stumps are decision trees consisting of just a single level and split [171]. A decision stump makes a prediction based on the value of just a single input feature, and is a good baseline classifier to compare against decision trees and other classifiers, to determine what results can already be achieved with a very basic model.
J48 Decision Tree (J48)	J48 is a WEKA implementation of the C.45 decision tree algorithm [178]. This algorithm generates a classification-decision tree for the given dataset by recursive partitioning of data.
k-Nearest Neighbour (k-NN)	k-NN classification finds a group of k objects in the training set that are most similar to the test object and bases its classification on the predominance of a particular class in this neighborhood [179].
Logistic Regression (LR)	LR uses a linear input combination of input variables to provide an output score, which is then mapped to a probability by applying a logistic function [61].
Naive Bayes (NB)	NB is a classification algorithm based on the Bayes rule of conditional probability. It assumes that the presence/absence of a particular feature of a class is unrelated (independence) to the presence/absence of any other feature [110].
Radial Basis Func. (RBF) Networks	RBF Networks are a type of feed-forward neural network. We used simple normalized Gaussian functions that each cover part of the input space and the activation of each of these functions given an output is then fed into a basic feed-forward neural network [120].
Random Forests	Random Forests is a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution of all trees in the forest [33].
Random Tree	The Random Tree algorithm builds a classification algorithm tree considering K randomly chosen predictors at each node. More explanation of Random Tree is provided in [101].

Table 2.3: *Confusion Matrix or Contingency Table*

Prediction Result		**Actual Result**
Y	**N**	
TP	*FN*	**Y**
FP	*TN*	**N**
Example:		
Y	**N**	
11	*33*	**Y**
10	*849*	**N**

Note :

True Positive (**TP**) : A positive instance that is correctly classified as positive
False Positive (**FP**) : A negative instance that is incorrectly classified as positive
True Negative (**TN**) : A negative instance that is correctly classified as negative
False Negative (**FN**) : A positive instance that is correctly classified as negative

Table 2.4: *Common Performance Measures and Terms*

Terms and Measures	**Description**
Overall Success Rate or Accuracy (*Acc*) [15]	$Acc = \frac{TP + TN}{TP + TN + FP + FN}$
True Positives Rate (*TPR*) or Recall or Sensitivity [58]	$TPR = \frac{TP}{TP + FN}$
False Positives Rate (*FPR*) [58] or false alarm ratio	$FPR = \frac{FP}{FP + TN}$
Precision	$Precision = \frac{TP}{TP + FP}$
F-measure (F_1) [26]	$F_1 = 2 * (\frac{Precision * Recall}{Precision + Recall})$

In this research, we search for algorithms that provide reliable estimates across the score range, thus we evaluate using the Area Under ROC[7] Curve (AUC) [79] value rather than accuracy. For imbalanced data, the AUC also avoids the issue of favouring models that just predict the majority outcome class. The larger the ROC area, the better the classification algorithm is in term of classifying classes [14]. The AUC value (calculated using WEKA) measures the performance of a model over the entire range of model scores, i.e. how well it separates by changing the score threshold of a class over the entire score range. Therefore, AUC shows the ability of the classification algorithms to rank classes correctly as more likely to be included in the class diagram or not. AUC is quite often be used to evaluate classification algorithms that utilized imbalanced dataset [39].

Precision and recall are common in information retrieval for evaluating classification performance [58]. However, these performance measures are not suitable to be used for our dataset (due to imbalanced or *class skew*). The F-measure aims to improve by balancing precision and recall, but the issue is that it still needs a fixed classification threshold[8] (in our case, there is no specified threshold as we aim to cover the whole range of scores). Therefore, AUC is preferred over accuracy, precision, recall and F-Measure (refer [62] for further discussion). The AUC measure is based on ROC graphs; A two-dimensional graph in which *TPR* is plotted on the Y-axis and *FPR* is plotted on the X-axis (see Figure 2.10 (a)). It indicates relative tradeoffs between true positives and false positives. Figure 2.10 [58] compares two classifiers evaluated using ROC curves and precision-recall curves. Figure 2.10 (a) and (b) show a balanced dataset (1:1 class distribution) while Figure 2.10 (c) and (d) show an imbalance dataset (1:10 class distributions[9] of the same classifier and same domain). This figure demonstrates that the ROC curves (Figure 2.10 (a) and (c)) are identical, but the precision-recall curves (Figure 2.10 (b) and (d)) differ substantially.

2.5 Summary

In this chapter, we defined the key concepts that are used in this research. We described the UML as our focus of this research. In particular, this research utilized the forward design and RE-CD as the main input. We use XMI as the input because it can be generated by most of the common CASE tools. Machine learning is the heart of this research as we use the classification algorithms to classify the classes that could be included and classes could be omitted in order to simplify the class diagram. The list of classified classes (included or excluded) from the classification process is meaningless if

[7] ROC means Receiver Operating Characteristics.

[8] A decision threshold is the cut-off degree employed to decide the final prediction of a classification model. In binary classification, the final prediction is class positive if the model's posterior probability of a test example is above the threshold; or else it is class negative [168].

[9] The classifier and the underlying concept are the same; different only in class distribution

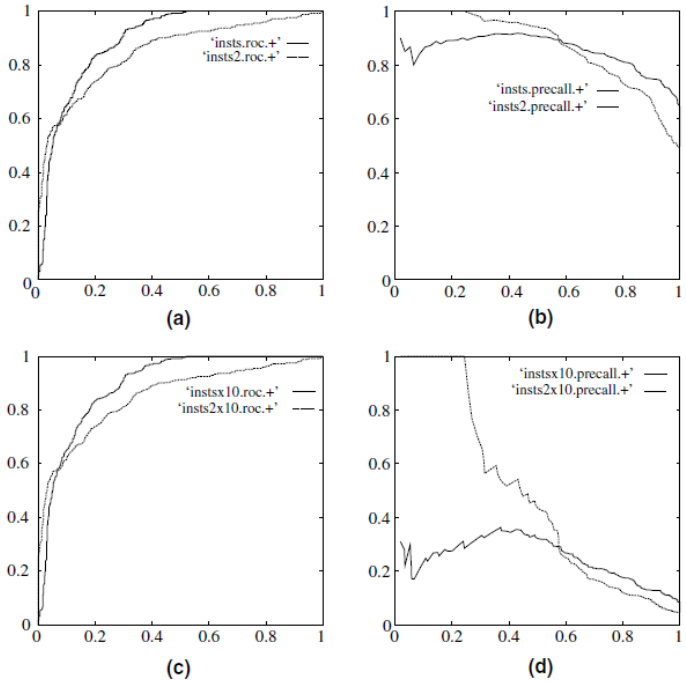


Figure 2.10: *ROC and Precision-Recall Curves under Class Skew*
(a) ROC curves (1:1 distribution) (b) precision-recall curves, (1:1 distribution)
(c) ROC curves, (1:10 distributions) (d) precision-recall curves, (1:10 distributions)

it is not presented graphically. Therefore, we refer to several cognitive design elements for software exploration that we believe useful to assist the software developer in understanding software.

This chapter only defines the common terms of this research. Other related terms and also related works are presented in each chapter.

Chapter 3

UML Usage in Open Source Software Development

UML is the standard for modeling software designs and is commonly used in commercial software development. However, little is known about the use of UML in Open Source Software Development. This chapter evaluates the usage of UML modeling in ten open-source projects selected from common open-source repositories. It covers the types of UML diagrams that are used, the level of detail that is applied, and the frequency of updating UML models. Our findings also include the application of UML modeling at different levels of detail for different purposes, the change in focus on types of diagram used over time, and research on how the size of models relates to the size of the implementation.

3.1 Introduction

UML provides the facility for software engineers to specify, construct, visualize and document the artifacts of a software-intensive system and to facilitate communication of ideas [32]. For commercial software development, the use of UML is commonly prescribed as part of a company-wide software development process while in open-source software development (OSSD), there is typically no mandate on the use of UML. Only if the community of developers of the OSSD feels the need (e.g. for their communication) then UML diagrams are produced. Even though some open-source projects employ UML diagrams, these diagrams do not completely correspond to the

This chapter is a more detailed version of a publication entitled **"UML Usage in Open Source Software Development : A Field Study"**, In Proceedings of the 3rd International Workshop on Experience and Empirical Studies in Software Modelling (EESSMod 2013)

implementation code. For instance, the number of classes used in class diagrams is typically less than the number of classes that exist in the implementation source code. The usage of UML class diagrams also varies across projects. Almost all OSSD projects that use UML choose to produce class diagrams. Some projects also constructed other types of UML diagrams such as use case diagrams, sequence diagrams and activity diagrams.

One of the benefits of UML is to ease communication between software developers. The nature of OSSD is that software developers normally communicate with each other using some online communication medium (e.g. discussion forum, e-mail, IRC) rather than through face-to-face interaction. There is an anecdotal belief that UML is rarely used in OSSD. However, there is no quantitative research to prove this perception. In this chapter, we aim at evaluating the usage of UML diagrams in OSSD projects. We want to investigate how UML is used in OSSD without the influence of the stakeholders or users of the system. We assume that the UML diagrams that exist in a project document means such diagrams are used in the project. The reason is that when the cost (effort) is spent in developing an artifact, such artifact should be used and provides a benefit in the development or maintenance phase [187].

We explore the publicly available software documentation to answer the following questions: 1) What types of UML diagrams are used? 2) How does the size of the design relate to the size of the implementation? 3) What level of detail is used in UML diagrams? and 4) How does timing of changes in the implementation relate to the changes in UML diagrams/documentation?

The chapter is structured as follows: Section 3.2 discusses related work. Section 3.3 describes the case studies used in this research. Section 3.4 explains the study approach while Section 3.5 presents the results and findings. This is followed by our conclusion and future work in Section 3.6.

3.2 Related Work

Dobing and Parsons [48] performed a survey to find out to what extent UML is used and for what purpose, what are the differences of the levels of detail used and how successful UML usage is for communication in a team. The survey was conducted using a web survey and participated by 171 UML practitioners. The research found that the most used types of UML diagrams were use case diagrams and class diagrams while collaboration diagrams were used the least. In [47], Dobing and Parsons also conducted another survey to investigate the current practice in the use of UML. There were 299 responses in the survey (with the endorsement of the Object Management Group (OMG) [119]). The findings of this survey highlighted that the most used UML diagrams were class diagrams, use case diagrams and sequence diagrams. This research also discovered that class diagrams and sequence diagrams play a major role in specifying system requirements for programmer, documenting the design for future

maintenance and in clarifying understanding of the application among team members.

Grossman et al. [67] performed a study on the individual perspective of using UML. This study also addressed the characteristics that affect the usage of UML. Similar to [47] and [48], the result of the most important diagrams in ranking are use case diagrams, class diagrams and sequence diagrams. Those studies also found out that it is difficult to determine whether UML provides too much detail or too little detail because it depends on the software technology (i.e. Enterprise System, Web-based system, real-time system). The study suggested that UML diagrams need to be customized based on the environment.

Yatani et al. [182] conducted an evaluation on the use of diagramming for communication among OSSD developers and also performed semi-structured interview with developers from a large OSSD project. This study highlighted a diverse types of diagrams that is used for the communication between the contributors of the system. Not all diagrams used for communication purposes were updated during the project.

Chung et al. [42] carried out a survey that was participated by 230 OSSD developers and designers. Their findings demonstrate that 1) In terms of frequency of updating designs, even though 76% agree that diagrams have value, only 27% practice diagramming very often or all the time, and 2) The UML diagrams are only used for formal documentation purposes.

Most of the related works use surveys to explore the usage of UML diagrams. These surveys are based on the practitioners' perspective of how they use UML. In contrast, our study evaluates the use of UML modeling in OSSD projects by mining the project documentation. Hence, this reflects the real artifacts produced by using the UML notation.

3.3 Case Study

One of the challenges of this study was to find suitable OSSD Projects that use UML diagrams. Based on research by Hutchinson et al. [81], Dobing and Parsons [48], and Erickson et al. [54], we know that one of the most used UML diagrams is the class diagram. For this reason, we performed a search for UML class diagram images using the Google[3] search engine. In particular, we targeted our search on four open source repositories: SourceForge[7], GoogleCode[4], GitHub[8] and BerliOS[2]. The primary keyword used for the search was "Class Diagram". Based on the hits of these searches, we browsed the project repositories to assess their suitability for inclusion in this study. Our initial list of candidate cases consisted of 57 projects (see Appendix A.1). We refined the selection of the case study by using the following criteria:

- The project should have UML diagrams and corresponding source code (projects that have multiple versions were preferred).
- The source code should be written in Java.

- The amount of classes (in the source code) > 50 classes.

The reason for selecting projects in Java was that we intended to reverse engineer the source code to class diagrams for analysis purposes. The reverse engineering tool that we used for this study performs best with Java source code. We refine our selection of case studies as follows:

1. *Round 1*: Out of 57 projects, we eliminate 21 projects that developed using C++, C#, Pascal, Phyton, etc. (other than Java). Only 36 projects remain to be the candidate.
2. *Round 2*: Discard 13 projects due to the number of classes below 50.
3. *Round 3*: Discard 6 projects because we prefer projects that have more than one version.
4. *Final Round*: In total, 18 projects qualify for the final round. In this round, we thoroughly explore the case studies artifacts (source code, class diagram, documentation). As a result, we found that ten projects that are suitable for our research. Most of the projects we discarded because the projects only provide the latest source code in the repository, even though the projects have several versions of releases (also class diagram).

The list of case studies is shown in Table 3.1. The total numbers of classes involved in these case studies range from 50 to 2000.

3.4 Approach

This section describes the approach we used in this study. We conducted four main activities in order to answer the following research questions:

RQ1: What types of UML diagrams are used?

Based on the project repository, we manually browsed the documentation and other provided information about the software to find all the UML diagrams that were used in the project.

RQ2: How does the size of the design relate to the size of the implementation?

Our aim was to use one single tool for counting classes of both the design and the implementation. Furthermore, for source code, we only wanted to count classes that were actually *designed* for the project's system, hence we exclude library classes (also test-classes) that are imported, and would typically not be modeled. To this end, source codes are reverse engineered (into class diagrams) using several CASE tools. The CASE tools used in this study were MagicDraw [9] version 17.0 and Enterprise Architect [153] version 7.5. The reverse engineered design was then exported to XML Metadata

Table 3.1: *List of Case Studies*

Project	Description	No. of Releases	URL Source
ArgoUML	An open source UML modeling tool and include support for all standard UML 1.4 diagrams.	19	http://argouml.sourceforge.net
Mars Simulation	Free software project to create a simulation of future human settlement of Mars.	26	http://mars-sim.sourceforge.net/
JavaClient	The project allows development of applications for Player/Stage using the Java programming language.	3	http://java-player.sourceforge.net/
JGAP	Genetic Algorithms and Genetic Programming package.	8	http://jgap.sourceforge.net/
Neuroph	Lightweight Java neural network framework to develop common neural network architectures.	9	http://neuroph.sourceforge.net/
JPMC	Java Portfolio Management Component (JPMC) is a collection of portfolio management components.	1	http://jpmc.sourceforge.net/
Wro4J	It stands for Web Resource Optimizer for Java. The project purpose is to improve web application page loading time.	3	http://code.google.com/p/wro4j/
xUML-Compiler (xUML)	xUml-Compiler takes a user specified data model and associated state machines and produces an executable and testable system.	13	http://code.google.com/p/xuml-compiler/
Maze	Maze-solver is a Micro-Mouse maze editor and simulator.	2	http://code.google.com/p/maze-solver/
Gwt-portlets	Free open source web framework for building GWT (Google Web Toolkit) applications.	6	http://code.google.com/p/gwt-portlets/

Interchange (XMI) files. These were loaded into a UML case tool in which we manually removed all library classes. From the resulting XMI files, software design metrics were computed using the SDMetrics [180] tool.

Table 3.2: *Levels of Detail in UML models*

No	Class Diagram Elements	Low LoD	High LoD
1	Classes (box and name)	YES	YES
2	Attributes	NO	YES
3	Types in Attributes	NO	YES
4	Operations	NO	YES
5	Parameters in Operations	NO	YES
6	Associations	YES	YES
7	Association Directionalities	NO	YES
8	Association Multiplicities	NO	YES
9	Aggregations	YES	YES
10	Compositions	YES	YES

RQ3: What level of detail is used in UML diagrams?

The level of detail (LoD) for all UML diagrams gathered from the projects' repositories was analysed using the level of detail that was defined by Fernández-Sáez et al. [60] (as illustrated in Table 3.2). In addition, we also analyzed the diagrams to identify the technique of constructing the UML diagram (forward or reverse engineering). The UML diagrams were identified as RE-CD if they satisfy the symptoms (or weaknesses) mentioned by Osman [125]. These tasks were done manually.

RQ4: How does timing of changes in implementation relate to the changes in UML diagrams/documentation?

For source code, we manually extracted the dates of releases from the project repositories. For UML diagrams, we looked at the date-information provided by the system documentation, developer's manual and other related documents in the project repository.

3.5 Results and Findings

This section describes the result of this study. The results are grouped by the research questions mentioned in the previous section.

3.5.1 Usage of UML Diagrams

The UML diagram that was mostly used in our set of OSSD projects is the class diagram. This was to be expected because our main keyword of searching for the case study was based on class diagrams. Table 3.3 shows which other types of diagrams were

Table 3.3: *UML Diagram Usage*

			Structure Diagram					Behaviour Diagram		
No	**Project**	**Use Case**	**Component Diagram**	**Package Diagram**	**Class Diagram**	**Composite Structure Diagram**	**Object Diagram**	**Sequence/ Interaction Diagram**	**Activity Diagram**	**State Machine Diagram**
1	Maze	No	No	No	Yes (6)	No	No	No	No	No
2	JavaClient	No	No	No	Yes (1)	No	No	No	No	No
3	xUML	No	No	No	Yes (1)	No	No	No	No	No
4	JPMC	Yes (1)	No	Yes (1)	Yes (4)	No	No	No	No	No
5	Neuroph	No	No	No	Yes (3)	No	No	No	No	No
6	Gwt-portlets	No	No	No	Yes (3)	No	No	Yes (1)	No	No
7	Wro4J	No	No	No	Yes (3)	No	No	No	No	No
8	JGAP	No	No	No	Yes (2)	No	No	No	No	No
9	ArgoUML	No	Yes (1)	Yes (12)	Yes (30)	No	No	Yes (2)	Yes (1)	No
10	Mars	No	Yes (2)	No	Yes (2)	No	No	No	Yes (1)	No
	Total no. of diagrams used (i.e. no. of '*yes*')	**1**	**2**	**2**	**10**	**0**	**0**	**2**	**2**	**0**
	Total no. of Diagram	**1**	**3**	**13**	**55**	**0**	**0**	**3**	**2**	**0**

used. The term 'Yes' in Table 3.3 means that the project used at least one instance of a UML diagram specified in the table. The numbers in Table 3.3 indicate the number of diagrams constructed in the OSSD projects. Similar to most industrial use, none of the OSSD projects used UML to model their complete system. The use of UML in OSSD projects seems driven by a need to codify high-level knowledge. For example, ArgoUML did not use sequence diagrams in their modeling until there was a new feature. Only this new feature was explained by a sequence diagram.

In general, the case studies showed that the most used UML diagrams in OSSD are use case, component, package, class, sequence/interaction and activity diagram. The following subsections describe the results in more detail.

Use Case Diagram

A use case diagram is used to describe the desired functionality of the software product [65]. Use case diagrams were used by only one of our evaluated OSSD project (see JPMC in Table 3.3). Most of these OSSD projects have specified their goal, but the specification and the interaction between users and system were explained in text.

Component Diagram

Component diagrams are used to divide the system into components and show their relationships through breakdown of components into lower-level structure [63]. This diagram is used to illustrate the high-level structure of large systems. Because of this

reason, only complex projects among the case studies used this diagram. ArgoUML and the Mars Simulation Project provided this diagram in their repositories. ArgoUML provided one component diagram from an old design document to illustrate the interaction between early developed component and packages. The Mars Simulation project provided two component diagrams, i.e. 'Top Level Diagram' and 'Simulation Component Diagram'. The 'Top Level Diagram' illustrated dependencies between 3 components while the 'Simulation Component Diagram' illustrated more details about the relationship between a simulation component and other related components.

Package Diagram

Package diagrams provide a grouping construct that allows to group design elements together into higher-level units [63]. Package diagrams show the relationship between higher level units. This diagram is used to explain the high-level structure of a system. Only two of the case studies used this diagram. The JPMC project presents almost all main packages and their dependencies in a package diagram. Meanwhile, ArgoUML presented two package diagrams. The first package diagram in this project illustrated the dependencies between domain-related packages and two other packages representing external libraries. The second package diagram illustrated the high-level package in this project.

Class Diagram

Class diagram is the most used UML diagram in these case studies. Most of the case studies only show classes that are important to the system. The correspondence between design classes and implementation is discussed in subsection 3.5.2.

Sequence/Interaction Diagram

Sequence diagrams were used by two of our evaluated OSSD projects. However, both projects have only one sequence diagram per project. ArgoUML introduced a sequence diagram after eight version of releases. Table 3.7 shows that only after version 0.26, a sequence diagram was introduced in the documentation. Perhaps, it is difficult to generate the sequence diagram for the entire release. Hence, the developer of this project used a sequence diagram to illustrate the flow of a new feature. The gwt-portlets project used only one sequence diagram. We assume that the described flow contains crucial information for the system. This is due to the classes that were involved in the sequence diagram were presented in the project's class diagram that shows the key classes of the system.

Activity Diagram

Activity diagramming or activity modeling emphasizes the flow and conditions for coordinating lower-level behaviour [68]. This study found that two OSSD projects used the activity diagram. However, not all activity diagrams in these projects were related to the software development. ArgoUML used an activity diagram to present the flow of managing issues in ArgoUML project. Meanwhile, the Mars Simulation project used one activity diagram for specifying a feature of the project.

3.5.2 Ratio between Design and Implementation

This subsection presents the results of analyzing the ratio between classes in the design and classes in the implementation. Since there are multiple versions of both the design and implementation in most of the case studies, we chose a pair with a high ratio of design to implementation. For example, for the Neuroph project we selected version 2.3 because this version has a high number of designed classes. The project starts updating UML diagrams at this point in time. The Maze project has the highest ratio of classes in design to classes in the implementation. This is a relatively small project that consisted of 69 classes in the implementation and 40% of these classes were represented in the UML design. In absolute numbers, the highest number of classes in a design was found in the JavaClient project with 57 classes. The data in Table 3.4 is depicted graphically in Figure 3.1. This figure shows that the ratio between the number of classes in the design and the number of classes in the implementation decreases when the number of classes in the implementation increases. Based on our observation, most of the projects had created UML diagrams in the early version of the release, but rarely increase the amount of classes in the design.

3.5.3 Level of Detail (LoD)

This subsection describes the result of assessing the level of detail used in modeling, as well as the use of reverse engineering in the case studies. The overall result is illustrated in Table 3.5. The project that uses the most class diagrams is ArgoUML: 46 class diagrams. Out of this total number, 19 UML diagrams were constructed in Low LoD and other 27 diagrams were constructed in High LoD. ArgoUML is the only case study that used RE-CDs. There were 15 diagrams which were constructed using reverse engineering techniques. Almost half of these diagrams were used to describe the user interface from an old design documentation and other diagrams showed class diagrams for selected classes. Most of the selected classes were classes that play a key role in how the program works [165]. The Maze project showed some interesting result in their construction of UML diagrams. There were six UML class diagrams constructed in this project. A class diagram with low LoD displays 40% of the total classes. These diagrams illustrate the relationships between domain-related classes

Table 3.4: *Classes in Design versus Classes in Implementation*

No	Project	# of Class Diagram	# of Classes in Design (D)	# of Classes in source code (S)	% D vs S
1	Maze	6	28	69	40.6
2	JavaClient	1	57	215	26.5
3	xUML	1	45	172	26.2
4	JPMC	4	24	126	19.1
5	Neuroph	3	24	179	13.4
6	gwt-Portlets	3	20	178	11.2
7	Wro4J	3	11	100	11.0
8	JGAP	2	18	191	9.4
9	ArgoUML	30	33	909	3.6
10	Mars Simulation	2	31	953	3.3
	Total	**55**	**291**	**3092**	**16.4**

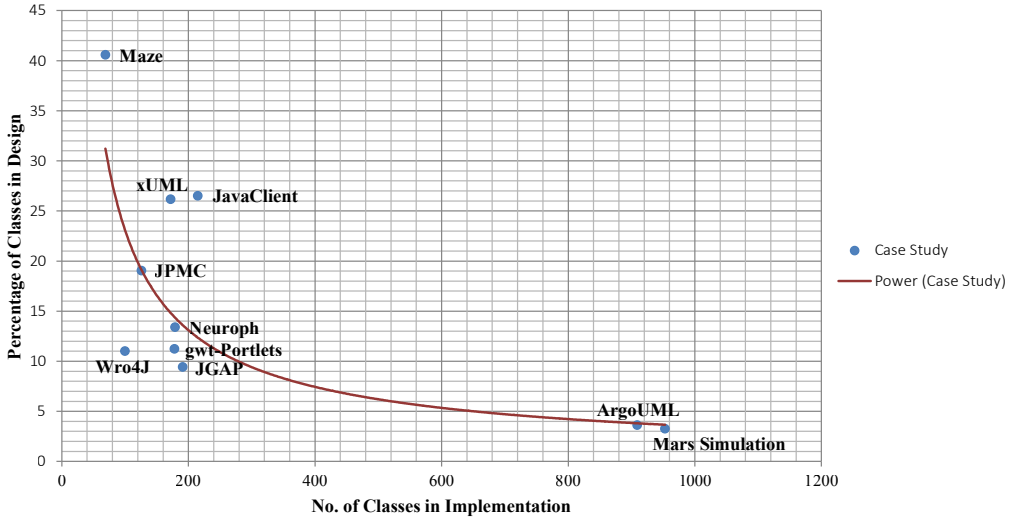


Figure 3.1: *Classes in Design vs Classes in Implementation*

Table 3.5: *LoD and Forward/Reverse Class Diagram*

No	Project	Low LoD	High LoD	Forward	Reverse	Total Class Diagram
1	**ArgoUML**	16	14	15	15	30
2	**Maze**	1	5	6	0	6
3	**JPMC**	0	4	4	0	4
4	**Mars**	2	0	2	0	2
5	**Wro4J**	2	1	3	0	3
6	**Neuroph**	1	2	3	0	3
7	**Gwt-Portlets**	1	2	3	0	3
8	**JGAP**	2	0	2	0	2
9	**Javaclient**	0	1	1	0	1
10	**xUML**	0	1	1	0	1
	Total	25	30	40	15	55

and external library classes used in this system. The other five diagrams presented class diagrams based on selected packages. We assume that the classes listed in the design play an important role in the system. In the Mars Simulation project, the activity diagram and the component diagram were constructed with in LoD.

The Wro4J project used only class diagrams. Two of the class diagrams were in Low LoD, while another class diagram was constructed in High LoD. This Low LoD class diagram was used to describe the structure of classes in the system. Similar to this project, Gwt-Portlets project also used a Low LoD class diagram to present the high-level class structure in the project. Another two class diagrams showed the classes and the relationship of important classes in the system. The Low LoD was also used to present the higher level of abstraction of the system class diagram in JGAP project. This project has two class diagrams from different versions of releases that showed the class diagrams of the key classes in the system.

In Neuroph, three class diagrams were presented in the system repository. Two of the diagrams were presented in High LoD. Those class diagrams were used to describe the class diagrams of important or key classes in the system. Meanwhile, all JPMC project's UML diagrams were constructed in High LoD. There were four class diagrams showing the key classes of the system. The high-level abstractions of class structures were presented in a package diagram and a component diagram.

The JavaClient and xUML-compiler project have only one High LoD diagram for each project. The JavaClient project constructed a very complex and high LoD class diagram that consisted of 67 classes. This class diagram consisted of classes that existed in the first version of this system. Hence, it is different from other case studies that normally showed the important, relevant or key classes in the system. The xUML-compiler project constructed a High LoD class diagram that show the relations between the domain-related classes and the external packages.

In addition to LoD, we also evaluate the usage of reverse engineering in OSSD projects. Initially, we expected the class diagrams that were constructed using reverse engineering to have a High LoD. However, we found several RE-CDs that had Low LoD in ArgoUML project. We found that only ArgoUML used reverse engineering for reconstructing some of its class diagrams. However, several UML diagrams from other projects also show several symptoms of RE-CDs (for example, "no aggregation and composition relationship" and "multiple relationships for the same direction"). Perhaps, these diagrams were constructed through reverse engineering, but were subsequently modified manually and ended up looking like a forward engineering design.

3.5.4 Frequency of Updating UML Models

This subsection presents the frequency of updating the UML models of the case studies. Basically, we would like to know whether UML diagrams are used throughout the projects or only in the initial phases. We analysed the case studies that have multiple versions of releases to assess the frequency of updating the diagrams while the systems evolve through subsequent releases. Even though there were multiple versions of system releases for the Mars Simulation, JavaClient, JPMC, Gwt-Portlet, Maze and xUML-compiler project, their UML diagrams were not changed. For instance, the Mars Simulation project has released 26 versions of source code. The UML designs were only uploaded on Dec 2009. Based on that date, we assume that this design corresponds with the release version 2.87 and above. This indicates that the earlier 19 versions of the software did not have a UML model. However, we could not disregard the fact that the design may be created earlier than the date it was uploaded.

The result also shows that the frequency of updating UML diagrams is low. In most of the case studies, a new UML diagram was created when there was a new feature of the system introduced in a new version or release. Only the Neuroph and ArgoUML project actually modified existing diagrams. Other projects only added new diagrams to their documentation, but did not modify previously existing diagrams. In the ArgoUML project, we found that there was an increasing amount of diagrams at the same time as the number of project contributors increased. The work by Wen Zhang et al. [186] shows that there was an increasing amount of participants in version 0.26. As we can see in Table 3.6, the ArgoUML project updated and added a lot of UML diagrams in version 0.26. We hypothesize that the documentation was elaborated to cater for a group of newcomer developers that was looking for information about the design. The creation of UML diagrams is perhaps being used to ease the communication of the new developers. It is also possible that the new software developers created this diagram to help their understanding of the system. In the next subsection, we discuss the ArgoUML project as an example of a project that did update their UML designs across multiple versions of releases.

Table 3.6: *Add, Remove and Modify of UML Diagrams in ArgoUML Project*

No	Release Version	UML Diagram			Remarks
		Add	Remove	Modify	
1	0.10.1	15	0	0	Old Document
2	0.12	18	0	0	Cookbook 2003 was added
3	0.14	0	0	0	Cookbook 0.14 was added but no changes for UML diagram
4	0.16	1	1	0	Cook book 0.16 was added. Key classes class diagram was taken out
5	0.18.1	0	0	0	Cookbook 0.18.1 was added but no changes for UML diagram
6	0.20	3	0	0	Cookbook 0.20 was added.
7	0.22	0	0	0	Cookbook 0.22 was added but no changes for UML diagram
8	0.24	1	0	0	Cookbook 0.24 was added
9	0.26	8	0	3	Cookbook 0.26 was added.
10	0.26.2	0	0	0	Cookbook 0.26.2 was added but no changes for UML diagram

ArgoUML

Table 3.7 shows which types of diagrams were used across subsequent versions over time. The table shows that in the early versions of the software, diagrams were made that represent the high-level structure of the system (component, package and class). As development time progresses, diagrams are added that represent the dynamic behaviour of the system through activity diagrams (v 0.16) and sequence diagrams (v 0.26). Also, at the later stages of development, component diagrams are no longer used. We believe this trend to be typical of the use of modeling in software development in general (for non-embedded applications): Firstly, the developers design the overall structure (using component diagrams) and later continue to flesh out using behavioural diagrams of the design. Figure 3.2 shows the evolution of UML diagrams in every version of release. Figure 3.2 also shows the evolution of the number of classes. It is explicitly shown that UML diagrams are rapidly created in the early stage of software release and then occasionally updated.

3.5.5 Key Classes

This study shows that UML diagrams do not cover the entire scope of the implementation. Class diagrams only show the key classes in the system. The OSSD developers identified the main classes of the system and showed these in a high-level class diagram. Perhaps, the package diagram is usually not used to present the high-level abstraction because it is too brief and a complete class diagram is not used because it contains too much information. In addition, we contacted one of the developers of the case studies. He confirmed that the developer in his project constructed UML diagrams only for important classes. Thus, we believed that there is a need for a class diagram abstraction or condensation method to produce this kind of diagram; as also suggested by Ichii et al. [82]. The studies performed by Andriyevska et al. [17] and Osman et al. [133] may be useful for the class diagram abstraction.

Table 3.7: *List of UML Diagrams used in ArgoUML Project*

No	Release Version	Date	Source	Component Diagram	Package Diagram	Class Diagram	Activity Diagram	Sequence/ Interaction Diagram
1	0.10.1	09.10.2002	Old Design Document	Yes	Yes	Yes	No	No
2	0.12	18.08.2003	Cookbook 2003 and Old Design Document	Yes	Yes	Yes	No	No
3	0.14	05.12.2003	Cookbook 2003 and Old Design Document	Yes	Yes	Yes	No	No
4	0.16	19.07.2004	Cookbook-0.16	No	Yes	Yes	Yes	No
5	0.18.1	30.04.2005	Cookbook-0.18.1	No	Yes	Yes	Yes	No
6	0.20	09.02.2006	Cookbook-0.20	No	Yes	Yes	Yes	No
7	0.22	08.08.2006	Cookbook-0.22	No	Yes	Yes	Yes	No
8	0.24	12.02.2007	Cookbook-0.24	No	Yes	Yes	Yes	No
9	0.26	27.09.2008	Cookbook-0.26	No	Yes	Yes	Yes	Yes
10	0.26.2	19.11.2008	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
11	0.28	23.03.2009	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
12	0.28.1	16.08.2009	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
13	0.30	06.03.2010	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
14	0.30.1	06.05.2010	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
15	0.30.2	08.07.2010	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
16	0.32	28.01.2011	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
17	0.32.1	23.02.2011	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
18	0.32.2	03.04.2011	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes
19	0.34	15.12.2011	Cookbook-0.26.2	No	Yes	Yes	Yes	Yes

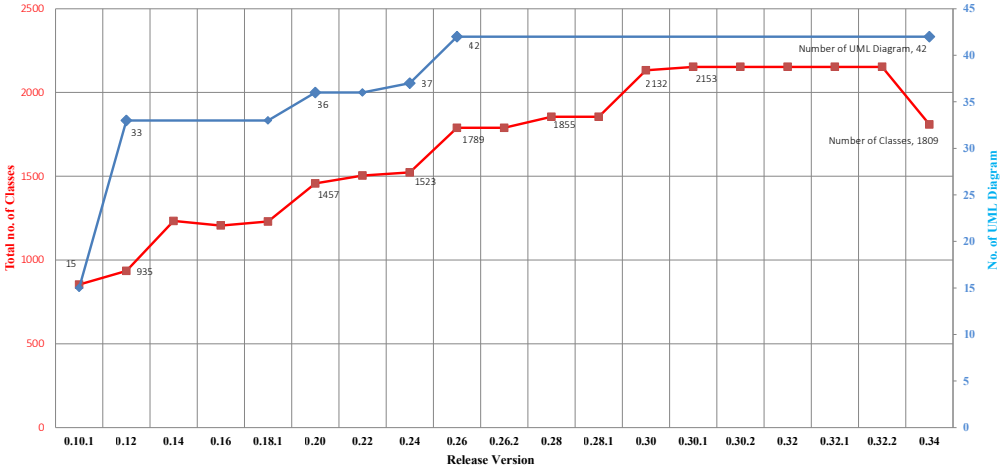


Figure 3.2: *ArgoUML Evolution in UML Diagrams and Number of Classes*

3.5.6 Threats to Validity

This section describes the threats to validity of this study. In terms of case study selection, there could be more case studies if we include more open source repositories and also include projects developed other than Java programming language. The selected projects may not be representative of all the OSSD because the selected case studies can be considered as small and medium type of system development and also specific to Java-based project. In addition, referring to figure 3.1, we also do not have projects with a number of classes between 250 and 800. The result could be different if more large projects are included in this study. We used the keywords of "Class Diagram" when we search for the suitable case study. We realized that there are possibility that we missed some projects that have UML diagrams, but stored in CASE tool's format (such as .zargo-ArgoUML and .uml-StarUML). The study was done based on using only the information in the project repositories and also the projects' websites. It may be the case that developers use UML in their communication or for internal use without uploading their diagrams in the project repository. This study also only uses the date listed as the upload date of the documents in the repositories. The document may be created far before the uploaded date. Thus, the matching of the date of documentation and the version may not be accurate.

3.6 Conclusion and Future Work

This study explored if UML diagrams are used in OSSD projects. To this end, ten case studies were collected from online repositories. Four main questions were studied: What types of UML diagrams are used? How does the ratio of the design relates to the size of the implementation? What level of detail is used in UML diagrams? and How does timing of changes in the implementation relate to the changes in UML diagrams/documentation? The main purpose of UML modeling in projects is to ease the communication between the developers. This also seems to apply to OSSD projects. UML diagrams (specifically class diagrams) with a low level of detail are used to show a high-level abstraction of the structure of the system. UML diagrams with a high-level of detail are used to elaborate key aspects of the design or complex aspects of the design. By studying the evolution of UML models across versions, we found that the focus of modeling shifts from structural aspects in the early phases of development, to dynamic behaviour in the later stages of development.

The frequency of updating UML models is low. We found two triggers for updating UML diagrams: 1) if there are changes in the features of the system, and 2) if there is a group of newcomers joining the project. The latter cause confirms the role of UML models as a way of codifying design knowledge for communicating the design. Overall, this chapter shows that open source projects can be used as empirical sources for studying the usage of UML modeling.

For future work, it would be interesting to extend this study by performing a broader survey or interview OSSD developers to find out the reasons for or against using UML diagrams in their development. Also, it is interesting to ask developers for their pattern in updating UML models. Furthermore, future work could be to find more case studies and to extend the case studies to languages other than Java. This would allow differentiating results between programming languages.

Chapter 4

Assessing the Correctness and Completeness of UML CASE tools in Reverse Engineering

This chapter focuses on Computer-aided Software Engineering (CASE) tools that offer functionality for reverse engineering into Unified Modeling Language (UML) models. Such tools can be used for design recovery and round-trip engineering. For these purposes, the quality and correctness of the reverse engineering capability of these tools are of key importance: Do the tools recover all important information from the source code? Are the reverse engineering results correct? What kind of information is presented in the result? Based on these questions, we compare eight UML CASE tools (six commercial tools and two open source tools). We evaluate i) the types of input that these tools can handle, ii) the types of diagrams that can be reconstructed, and iii) the quality of resulting diagrams.

4.1 Introduction

The Unified Modeling Language (UML) is the standard for graphically representing the design of object-oriented software systems. While UML diagrams are created in forward design, these diagrams are poorly maintained. Maintaining correspondence

This chapter is adapted from publications entitled **"An Assessment of Reverse Engineering Capabilities of UML Case Tools"**, In Proceedings of the 2nd Annual International Conference of Software Engineering & Applications (SEA 2011) and **"Correctness and Completeness of CASE tools in Reverse Engineering Source Code into UML Model"**, In GSTF Journal on Computing vol.2, num.1 (2012)

(between design and implementation) is particularly challenging because over time an implementation tends to evolve considerably from its initial design [118]. Design models produced during the design phase are often forgotten during the implementation phase-under time pressure usually–and thus, present major discrepancies with their actual implementation are frequently present [72]. Lethbridge et al. [103] confirm the widely held belief that software engineers typically do not update the documentation as timely or completely as software process personnel and managers advocate. Tools support during maintenance, re-engineering or re-architecting activities has become important to decrease the time software personnel spend on manual source code analysis and help to focus attention on important program understanding issues [97].

Nowadays, a lot of commercial and open source CASE tools support reverse engineering. These tools provide the capability in reconstructing package and class diagrams based on source code, objects and/or executable files. These tools also provide an automated and semi-automated analysis of the software system regarding the software structure such as class, attribute and operation. Some of the CASE tools extend the UML reverse engineering capabilities by supporting sequence diagrams reconstruction (based on static analysis).

For this research, our motivation is to discover to what extent the CASE tools are able to reverse engineer UML diagrams out of source code. This information is useful because RE-CDs is one of the important inputs in this research. Particularly in UML class diagrams, we want to know what type of information provided in the RE-CD compared to the typical information that may exist in class diagrams (domain model/forward design). The study of this chapter also aim to gather information about CASE tool(s) that are suitable for the research in this thesis. In order to find the answers, we examined and compared the reverse engineering capabilities provided by the CASE tools. In total, eight CASE tools have been selected in this study, namely Visual Paradigm, Rational Software Architect, StarUML, Altova UModel, MyEclipse, Enterprise Architect, MagicDraw and ArgoUML. To understand how the tools analyze class diagram, we conduct three experiments.

The first experiment aims at discovering the capability of the evaluated CASE tools in performing the round-trip engineering [80] task. The second experiment evaluates the tools' capabilities of identifying class relationships (association, aggregation and composition) based on the code stated in [72]. The third experiment assesses the correctness and completeness of the tools in reverse engineering source codes into class diagrams.

The chapter is structured as follows. Section 4.2 presents the related work. Section 4.3 briefly describes the examined tools and properties used in this evaluation. Section 4.4 describes the sample cases and Section 4.5 explains the approach of this experiment. Section 4.6 presents our results and findings. Our evaluation is discussed in Section 4.7. This is followed by our conclusion and future work in Section 4.8.

4.2 Related Work

This section discusses work that related to this study. The following researchers have conducted several evaluations and comparisons of reverse engineering tools.

Kollmann et al. [94] presented a study that examined the reverse engineering capabilities of two CASE tools (Rational Rose, Borland Together), and compared the result with two academic prototypes (Fujaba, IDEA). Their study investigates the strengths, weaknesses and similarities of the tools' capability. In our research, we examine six commercial CASE tools and two open source CASE tools that we believe are commonly used in the industry. We extend the examination by observing the capabilities of the tools in reverse engineering the source code into package diagram and sequence diagram.

Koskinen and Lehmonen [97] analysedten reverse engineering tools in term of four aspects: data structures, visualization mechanisms, information request specification mechanisms and navigation features. Their research focused on the information retrieval capabilities of the selected tools. However, not all of their selected tools were capable of reconstructing UML diagrams. In our study, we selected tools that support reconstruction of UML models.

Bellay and Gall [22] presented a study that compared reverse engineering tools for the C programming language. Four reverse engineering tools were selected in the study. Their study aimed to discover the strength and weakness of the selected reverse engineering tools based on their usability, extensibility and applicability for embedded software systems. The tools selected in their study were different in functionality and capability. In contrast, our evaluated tools are comparable because the functionalities of the evaluated tools are relatively similar.

Gahalaut and Khandnor [64] presented a study about reverse engineering Java code. The study aimed to compare bytecode reverse engineering tools (decompiler) with UML reverse engineering tools (Altova UModel and Enterprise Architect). The inputs for this comparison were Java source code and Java class files. They stated that the decompiler and the UML reverse engineering tools generated the same class structures. However, our extended study found that although the structure were about the same, the detail in class information and the relationship were different if we compare RE-CDs that are constructed based on the class file and Java source file.

Akehurst et al. [13] focused on providing solutions to the issues of mapping qualified associations and the UML 2.0 semantic variations of an association into Java 5. They presented a comparison of forward engineering functionality of some CASE tools. In contrast, our evaluation covers forward and reverse engineering of class diagrams based on the user's view. Their study was centered on how to generate code based on the design. Our study evaluates and compares the tools' capabilities to reverse engineer basic class information and relationships.

Boklund et al. [30] performed a comparative study of forward and reverse engi-

neering in UML tools; focused on three-tier layered web services application. They evaluated four modeling tools in the perspective of UML-Modeling, UML-based Code Generation and Reverse Engineering UML-diagram from code. However, not all tools that they have selected can be used in their evaluation. In contrast, our selected tools are comparable in term of the tools capability and functionality.

Kearney and Power [87] proposed a framework and automated tool for benchmarking UML CASE tools reverse engineering capabilities. The automated tools presented in this study tightly rely on the input from software metrics tools. Although we conduct our experiment semi-automated, we present more information rather than concentrate only on software metrics.

4.3 Examined Tools and Properties

This section describes the examined tools and properties that are involved in this experiment.

4.3.1 Examined Tools

The CASE tools were chosen based on the following criteria:

- Capable of performing forward and reverse engineering in Java.
- Capable of exporting UML Model to XML Metadata Interchange (XMI) format.

In total, eight well-known CASE tools are selected as listed in Table 4.1. For commercial CASE tools, we used fully functional evaluation and academic evaluation versions. We used SDMetrics [180] (version 2.11 – academic license) to extract UML model information from different versions and different type of XMI files.

4.3.2 Examined Properties

The examined properties are divided into two parts: Reverse Engineering Capability and Class Diagram Properties.

Reverse Engineering Capability

The reverse engineering tool's capabilities are evaluated from three perspectives: UML diagrams, supported programming languages and supported types of source.

- *UML Diagrams*
  The selected types of UML diagram are: package diagram, class diagram and sequence diagram. We analysed all selected diagrams by evaluating (1) the process of reconstructing (reverse engineer) the diagrams, and (2) the output in term of completeness and representation. Only static analysis is used for reconstructing those diagrams.

Table 4.1: *List of Evaluated CASE Tools*

No	CASE Tool	Information	Vendor	License Type
1	Visual Paradigm 8.1	http://www.visual-paradigm.com	Visual Paradigm	Evaluation
2	MagicDraw 17.0	http://www.magicdraw.com	No Magic	Evaluation (academic)
3	Altova Umodel 2011	http://www.altova.com	Altova	Evaluation
4	Enterprise Architect 8.0	http://www.sparxsystems.com.au	Sparx System	Evaluation
5	Rational Software Architect 8.0.1	http://www-142.ibm.com/software/products/my/en/swarchitect-websphere	IBM	Evaluation
6	MyEclipse 8.6	http://www.myeclipseide.com	Genuitec	Evaluation (academic)
7	StarUML 5	http://staruml.sourceforge.net	StarUML	Open Source
8	ArgoUML	http://argouml.tigris.org	Tigris.org	Open Source

- *Supported Programming Languages*
  We study the capability of the CASE tools to reverse engineer source code from several common programming languages: PHP5, C++, Java, C#, Delphi, Python and Visual Basic (V.B). However, we perform reverse engineering using a sample case developed in Java. Other programming languages are evaluated based on the documentation/manual provided by the CASE tools' provider.
- *Additional Types of Input Formats*
  The supported input-types for reverse engineering UML diagrams (in addition to source code; e.g. binaries).

Class Diagram Properties

We evaluate the class diagram properties based on the following elements:

- *Attributes and Methods*
  - Number of attributes: The tools' ability to reconstruct all attributes including the type of attribute (public, private, protected) defined in the source code.
  - Number of operations: The tools' ability to reconstruct all methods (of all: public, private, protected, constructor) defined in the source code.
- *Relationship*
  - Number and types of relationship: The ability of the tools to reconstruct all relationships between classes.

– Association relationship: The capability of the tool in detecting association and binary association [68] relationship (i.e. aggregation[1] and composition[2]).

4.4 Sample Cases

This section describes the sample of cases that are used in this evaluation.

4.4.1 Movie Catalog System (MovieCat)

This sample case is derived from [177]. We altered the relationships in this class diagram to make sure all types of relationship were presented. This sample case is selected because of a little amount of classes and the class diagram can be altered to suit our purpose in this research. We use this sample case to evaluate the class diagram properties.

4.4.2 Automatic Teller Machine (ATM) Simulation System

This sample case is selected because it has a fully functional simulation system, a forward design and a complete implementation source code. It is developed by the Department of Mathematics and Computer Science, Gordon College [28]. The complete software documents consist of 22 designed classes. Various types of relationships were used in the UML diagram such as association, composition, dependency and inheritance. Some of the elements (especially class relationship) in this sample case have been altered to suit our requirement for the experiment. This sample case was used to evaluate the reverse engineering capability.

4.5 Approach

This section explains our approach to evaluating the tools. The evaluation is divided into two parts which are: Round-trip Capability and Reconstruction of UML Diagram types.

4.5.1 Round-trip Capability

We performed the round-trip capability experiment to assess the completeness of the CASE tools in recovering all information specified in the forward design (illustrated

[1] Aggregation (or shared aggregation) is a part-of relationship [63]. It is used when part of an instance (or class) is independent which means, if the related instance is deleted, the other instance may still exist.

[2] Composition (or composite aggregation) is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it [68].

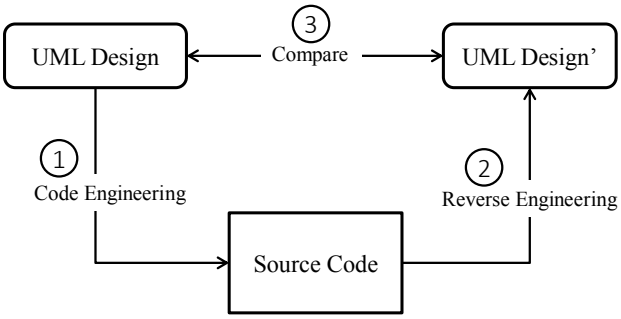


Figure 4.1: *Round-trip Engineering Experiment*

in Figure 4.1). We expected to get an overview of an automation of software lifecycle phases (i.e. software design->code generation->reverse engineering) using CASE tools. This experiment begins with creating the forward design class diagram (UML Design) that consisted of: (i) Attributes and Methods (private, protected, public) and, (ii) Relationships (association, aggregation, composition, inheritance). We generate the source code based on the UML Design through "Code Engineering" process, and produce the UML Design' by reverse engineer this source code. Then, we compare the UML Design and UML Design'.

4.5.2 Reconstruction of UML Diagram Types (package/class/sequence)

To assess the capability and the quality of the reverse engineering of UML diagrams, we conduct the following experiments.

UML Diagrams Reconstruction Capability

We evaluate the supported types of (reverse engineered) diagrams by i) reconstructing the diagram using the CASE tools and/or ii) finding the information on the tools' documentation/manuals. The tools' capabilities of reconstructing UML diagrams are analysed using a three-level scale which are described as follows:

- "+" : The tool is able to reverse engineer the specified diagram.
- "o" : The tool is able to reverse engineer the specified diagram, but present minimal information. For instance, the CASE tool is capable of presenting classes, attributes and operations but not relationships. Another example is the tool needed user intervention to generate the sequence diagram.
- "-" : The tool is unable to reverse engineer the specified diagram.

The information about the supported programming language and supported types of reverse engineering sources are gathered from the tool's documentation or/and manuals.

Detection of Aggregation, Association and Composition Relationship

The evaluation of reconstructing various types of class relationship is performed by using the source code defined in [72]. We created a source code that represents each evaluated relationship (i.e. aggregation, association and composition). Then, we reverse engineer this source code to evaluate the ability of the CASE tools in detecting multiple types of relationship.

Correctness and Completeness (CnC) of Reconstructed UML Diagram

This experiment aims at evaluating the completeness and correctness of the RE-CD constructed by the CASE tools. For the expected result (concrete result), we manually extract all class diagram information from the sample case design document and implementation code. For instance, we calculate (manually) the number of attribute and operation in every class in the class diagram. Then, the class diagram information gathered from RE-CDs (from each tool) are compared with the expected result. The evaluation is divided into two parts:

- *CnC of Class Information*: We tested all possible options to reconstruct the best RE-CD for each tool. The diagrams then exported to XMI files. We extract the metrics from the XMI files and compare the class diagram information with our expected result.
- *CnC of Reconstruction of Class Relationship*: All possible reverse engineering options are evaluated to achieve the best view of class relationship. Then, we manually compare the RE-CD relationships with our expected result.

4.6 Result and Findings

In this section, we present our findings which include: Reverse Engineering Capability and Class Diagram Properties.

4.6.1 Reverse Engineering Capability

In this subsection, we present the capability of the CASE tools in reconstructing the UML Diagrams, the supported programming languages and the supported types of source.

UML Diagrams

The results in Table 4.2 show that most of the tools were capable of reconstructing package diagrams. Visual Paradigm, MagicDraw and Altova UModel are good at reconstruction package diagram because these tools can perform this task automatically. An example of a reverse engineered package diagram is shown in Figure 4.2.

Table 4.2: *Supported UML Diagrams for Reverse Engineering*

No	Tools	UML Diagram		
		Package	**Class**	**Sequence**
1	Visual Paradigm 8.1	+	+	o
2	MagicDraw 17.0	+	+	o
3	Altova Umodel 2011	+	+	-
4	Enterprise Architect 8.0	o	+	-
5	Rational Software Architect 8.0.1	o	+	-
6	MyEclipse 8.6	o	+	o
7	StarUML 5	o	+	o
8	ArgoUML	o	o	-

In terms of the class diagram, all the evaluated tools are good at automatically reconstructing RE-CD (from source code) except ArgoUML. The reason is that the ArgoUML was unable to reconstruct the class relationship other than inheritance. All CASE tools give an option to generate the class diagram separately using the "drag and drop" function.

The reconstruction of sequence diagrams requires a lot of manual intervention because the users need to provide the information on the methods and classes that should be in the sequence diagram. Based on our experiment, only four tools have the capability of reverse engineering sequence diagrams. An example of a reverse engineered sequence diagram is illustrated in Figure 4.3.

The Supported Programming Languages

The supported programming languages results are presented in Table 4.3. It shows that the Enterprise Architect is able to reverse engineer all the programming languages listed in this evaluation. We also found that all evaluated tools were able to reverse engineer source code in Java.

The Types of Source

Overall, all evaluated CASE tools can reverse engineer class diagrams out of source code files (e.g. .java, .cpp and .cs). The CASE tools also offer an option to specify the source directory, and then automatically determine the source code file from the directory. Visual Paradigm, Altova and Enterprise Architect are capable of decompiling and reconstructing class diagram based on Java bytecode (.class), dynamic link library (.dll), execution file (.exe) and Java archive (.jar). Then, the tools generate class information that enable the users to construct a class diagram. The supported types of source are presented in Table 4.4.

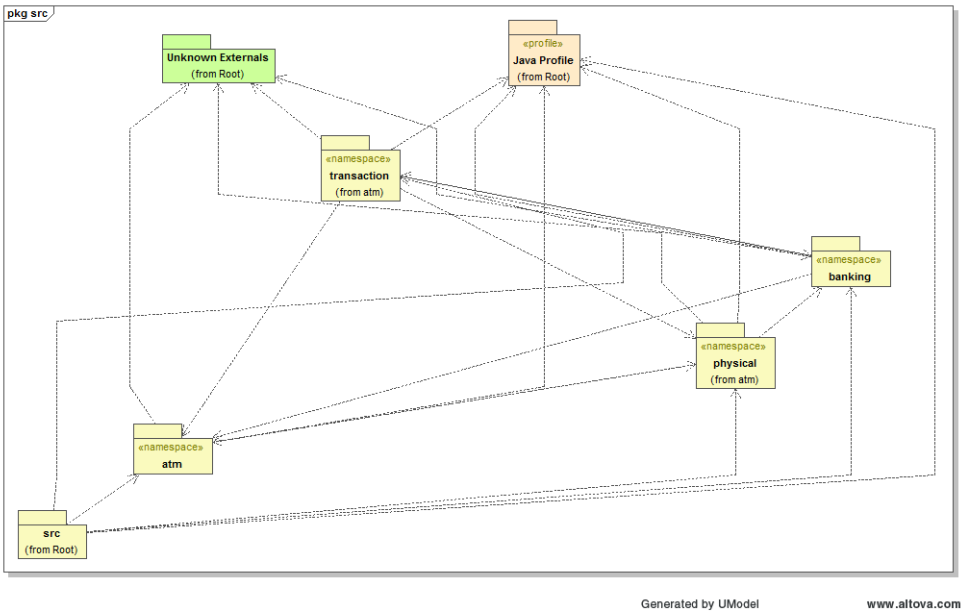


Figure 4.2: *Altova Reverse Engineered Package Diagram*

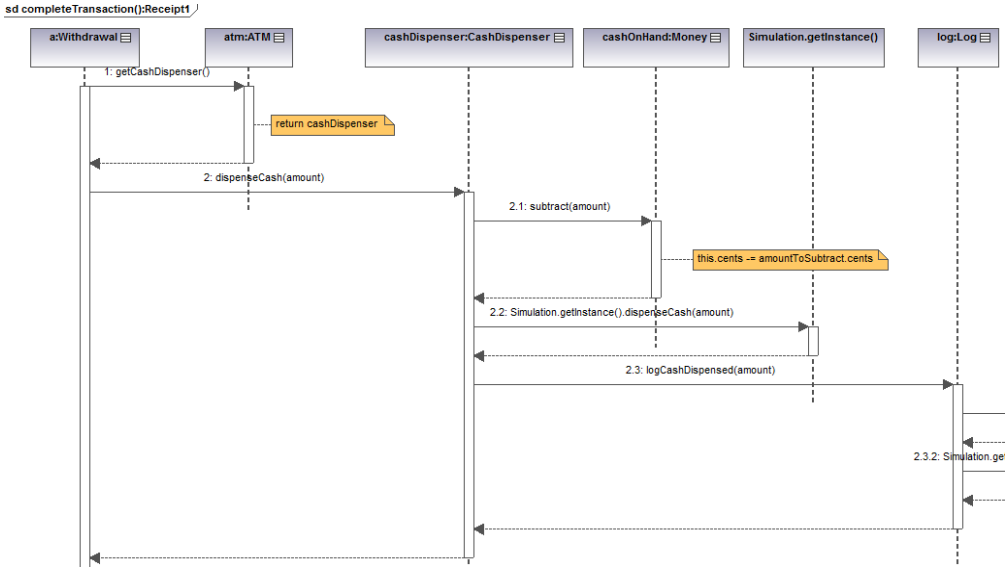


Figure 4.3: *Reverse Engineered Sequence Diagram using Altova*

Table 4.3: *Supported Programming Language for Reverse Engineering*

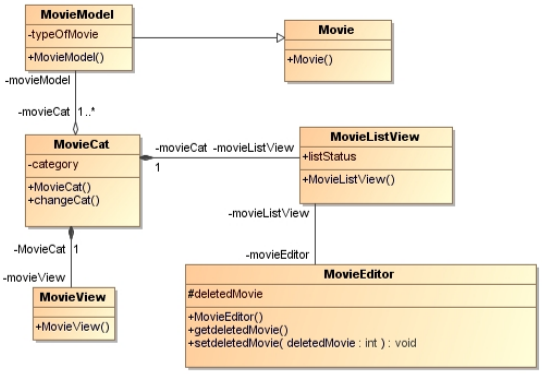
No	Tools	PHP 5	C++	Java	Delphi	Phyton	V.B	C#	Total 'Y'
1	Visual Paradigm	Y	Y	Y	N	Y	N	N	4
2	Altova UModel	N	N	Y	N	N	Y	Y	3
3	MyEclipse	N	N	Y	N	N	N	N	1
4	StarUML	N	Y	Y	N	N	N	Y	3
5	MagicDraw	N	Y	Y	N	N	N	Y	3
6	Rational Software Architect	N	Y	Y	N	N	Y	Y	4
7	Enterprise Architect	Y	Y	Y	Y	Y	Y	Y	7
8	ArgoUML	N	Y	Y	N	N	N	Y	3

4.6.2 Class Diagram Properties

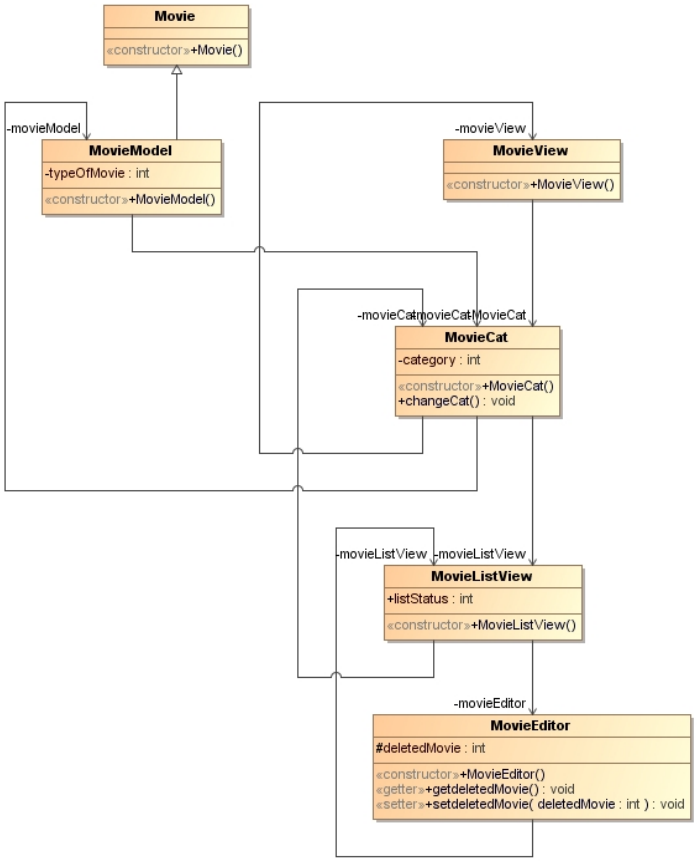
This subsection presents the assessment of the class diagram properties. The results are divided into: Round-trip Findings, Class Relationship Test, and Class Diagram Correctness and Completeness.

Round-trip Findings

The CASE tools successfully round-trip the information for class attributes and operations. However, the round-trip result for class relationships is different amongst the CASE tools. In general, all CASE tools correctly round-trip association and inheritance relationship. The aggregation and composition relationships were presented as association. Only Rational Software Architect presented aggregation and composition as a dependency. These aggregation and composition relationships were failed to be detected during the round-trip experiment due to the code-engineering process (transform design to source code) that defined those relationships as a link declaration [177] in the source code. Therefore, the tools cannot differentiate the types of relationship (example is in Figure 4.4). The discontinuity that may exist between object-oriented modeling language and programming language may be the reason behind this result. This discontinuity arises from the ambiguous concept in modeling languages and lack of corresponding concepts in programming languages [72]. These relationships represent different knowledge in software design. Unable to recognize these relationships correctly may hinder the traceability between source code and design, hence, obstructing software analysis.



(a) Forward Engineering Class Diagram



(b) Reverse Engineered Class Diagram

Figure 4.4: *Round-trip Test Result*

Table 4.4: *Additional Types of Input Format*

No	Tools	Source Code	Class/Object /Dynamic Link Library	Executable	Other
1	Visual Paradigm	.java, .cpp, .h, .php	.dll, .class, .inc	.exe, .jar	Source Directory, .zip
2	AltovaUModel	.java	.dll, Global Cache (GAC), MSVS .Net, .class	.exe, .jar	Source Directory
3	MyEclipse	.java	-	-	Source Directory
4	StarUML	.java, .cpp, .h, .cs	-	-	Source Directory
5	MagicDraw	.java, .cpp, .h, .cc, .cs	-	-	Source Directory
6	Rational Software Architect	.java, .cpp, .h, .cc	-	-	Source Directory
7	Enterprise Architect	.java, .h, .cs, .hpp, .pas, .php, .php4, .inc, .py, .vb, .cls, .frm, .ctl	.class, .dll	.exe, .jar	Source Directory
8	ArgoUML	.java, .cpp, .cs	.class	.jar	Source Directory

Class Relationship Test

The results of this experiment are illustrated in Table 4.5. It shows that all CASE tools are unable to reconstruct the specified relationships based on the source code (defined in [72]). Visual Paradigm, Altova UModel, StarUML, MyEclipse, MagicDraw and Enterprise Architect unable to generate the association relationships, while the aggregation and composition relationships were presented as association relationships. An example of aggregation test results is shown in Figure 4.5.

Class Diagram Correctness and Completeness

Class Diagram Correctness and Completeness (CnC) evaluation is divided into two parts: Attributes and Methods, and Relationship.

Table 4.5: *Class Relationship Test Result*

No	Tools	Association	Aggregation	Composition
1.	Visual Paradigm	No relationship presented	Present as association	Present as association
2.	Altova UModel	No relationship presented	Present as association	Present as association
3.	MyEclipse	No relationship presented	Present as association	Present as association
4.	StarUML	No relationship presented	Present as association	Present as association
5.	MagicDraw	Present as dependency	Present as association and dependency	Present as association and dependency
6.	Rational Software Architect	Present as dependency	Present as dependency	Present as dependency
7.	Enterprise Architect	No relationship presented	Present as association	Present as association
8.	ArgoUML	No relationship presented	No relationship presented	No relationship presented

1. *CnC of Attributes and Methods* evaluation presents the capability of the CASE tools in identifying class attributes and methods (or operations). The results are presented in Figure 4.6 and the explanations are the following:

    - **Number of Attribute(NA)**: We expected the tools to extract 79 attributes (NA) from the sample case. Visual Paradigm, Enterprise Architect and Rational Software Architect successfully extracted all the attributes. However, Rational Software Architect can only show the attributes by using the "Drag and Drop" function instead of using the "Transform" function. The class diagram generated from the "Transform" function did not show any attribute even though it exist in the tools "Project Explorer" pane. Furthermore, the generated XMI file also does not include any attribute. Other tools like Altova UModel, MyEclipse, StarUML and MagicDraw were unable to extract all the expected attributes.
    - **Number of Operations(NO)**: We expected the tools to extract 91 operations. However, most of the tools found more than expected. The reason is that the additional operations come from the "superclass" operations. StarUML did not completely extract all operations because it was unable to extract 4 constructors of 4 classes. On the other hand, Visual Paradigm extracted 77 operations.

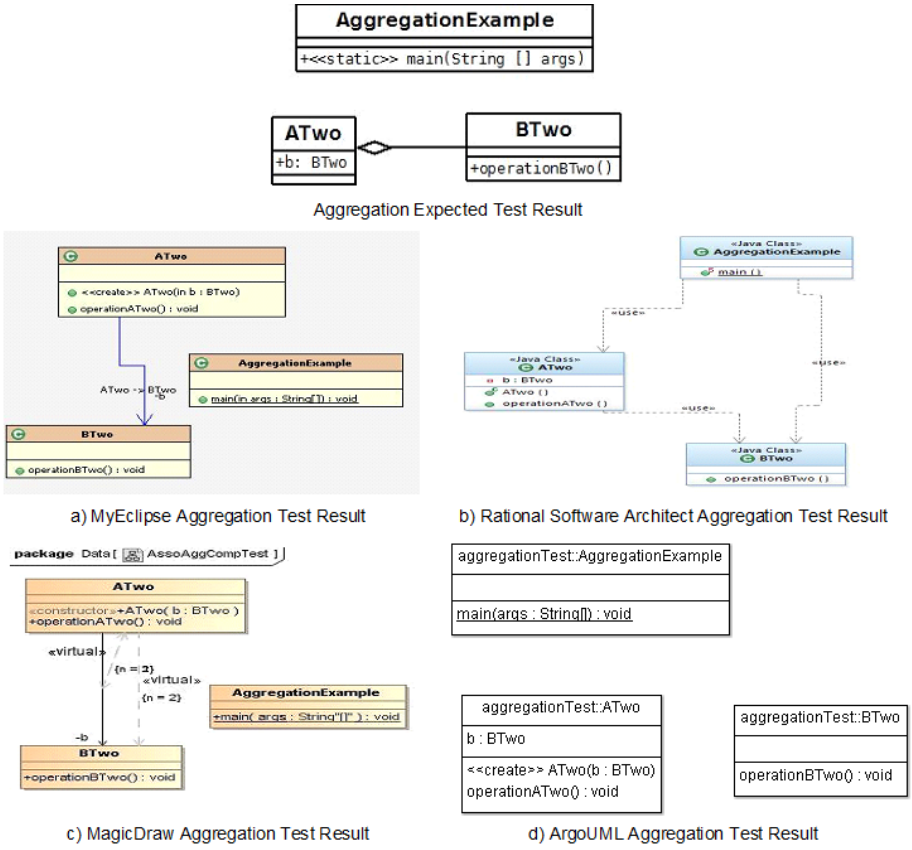


Figure 4.5: *Example of Diagram on Aggregation Test*

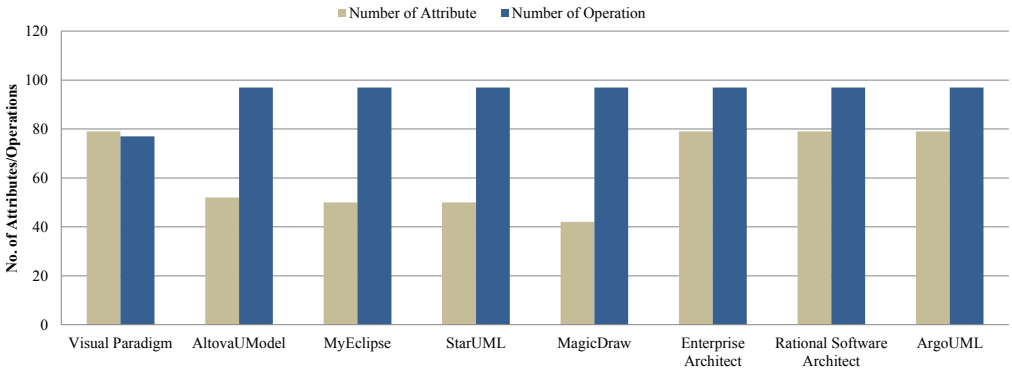


Figure 4.6: *Number of Attributes and Methods*

Table 4.6: *Relationship Correctness*

No	Tools	No. of Relationship	Association		Inheritance	
			Total	Correctness (%)	Total	Correctness (%)
1	Visual Paradigm	31	27	54.05	4	100
2	StarUML	31	27	54.05	4	100
3	Enterprise Architect	31	27	54.05	4	100
4	Rational Software Architect	30	26	67.57	4	100
5	MagicDraw	31	27	54.05	4	100
6	MyEclipse	20	16	27.03	4	100
7	Altova UModel	31	27	54.05	4	100
8	ArgoUML	4	0	0	4	100

2. *CnC of Class Relationship* evaluates the tools' capability of extracting association and generalization (inheritance) relationships. Other class relationships are not included in this evaluation as the Round-trip results (see Section 4.6.2) shows that the evaluated tools can only identify these relationships. For this purpose, we extracted all link declarations from our sample case (ATM simulation system) and used it as the expected result. In total, there are 41 relationships identified that consist of 37 association relationships and 4 generalization relationships. Of these relationships, three of them were bidirectional. The overall results are presented in Table 4.6. We found that only Rational Software Architect was capable of reconstructing bidirectional relationship. Other tools (except ArgoUML) reconstruct bidirectional relations by means of two separate links in opposite directions (example in Figure 4.7). Rational Software Architect also capable of presenting two separated relationships that directed to the same class as a single relationship. Other tools identified this kind of relationships as two separated associations.

4.7 Discussion

This section discusses the experiment results.

- **Strength** : Most of the tools are excellent in recovering the class attributes and methods. From the result, the tools were capable of extracting source code, visu-

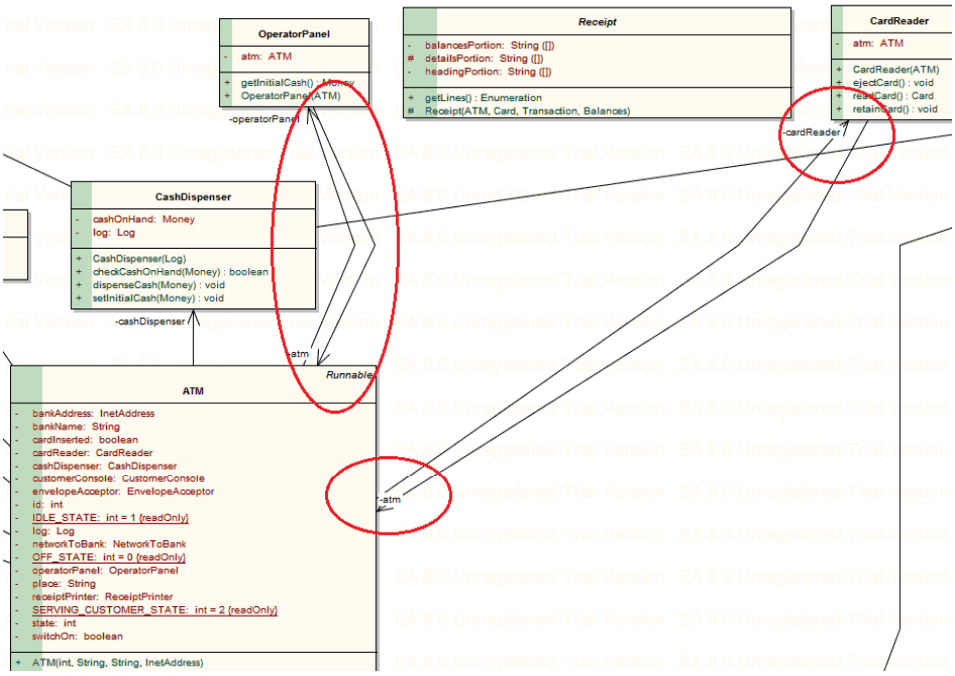


Figure 4.7: *Bidirectional Relationship with Two Separate Links*

alizing the class diagram and enabling manipulation of the generated diagrams. Some of the tools such as Altova UModel and Visual Paradigm are able to generate the class diagram automatically; most of the tools need user intervention to drag and drop the classes in the project explorer-canvas to recreate a class diagram. This drag and drop function can be useful to the user to select the reverse engineered classes that they desired to visualize in the class diagram. Of course, user intervention requires additional effort.

- **Weakness**: All CASE tools are unable to identify all the class relationships correctly. Most of the tools identify aggregation and composition relationships as association relationships. Rational Software Architect has presented the result differently by presenting dependency relationships for all class relationships that were tested. For further investigation, we tested all evaluated tools by generating the source code based on the design and then we reverse engineered the generated source code to produce the design (Round-trip). As the result, this test indicated that we were unable to generate the same design that we created. We observed the generated source code and it showed that the tools did not differentiate code generation between those types of relationship. This may be the reason of the tools are unable to recover the class relationship correctly. Incorrect code generation would lead to inaccurate synchronization between source code

and software design document. The design document becomes out-of-sync that decays the knowledge of the document.
The tools' weaknesses in generating code (forward engineering) and reverse engineer source code for class relationship have mentioned by Ralf Kollmann et al. (2002) [94] and Akehurst et al. (2007) [13]. Although this study is more recent, the tools are still unable to generate correct class relationships. However, two tools (MagicDraw, Rational Software Architect) give additional information by presenting dependency relationships as an addition to class relationship (association, aggregation and composition). The aggregation and composition relationship are essential to show how the software works. This relationship information may give some hints for the software engineer or the software maintainer which classes are important based on the software design before they browse the source code. The class relationship knowledge (especially which class to initialize after another) has to be discovered before the software engineer or software maintainer touch the source code.

Today, CASE tools support the reverse engineering capability not only by using source code as input but also support object or class files and executable files such as .jar and .exe. Some tools such as Altova UModel, Rational Software Architect and Visual Paradigm offer more functionality where they are able to present sequence diagrams based on the reverse engineering result. Although they are not able to automatically generate the sequence diagram, it at least may help the software engineer or the software maintainer to understand the class interactions.

Overall, from the user point of view, the functionality to do reverse and forward engineering are easy to access by the user and the tools give good instruction and information to the user to use the functionality and analyze the results.

The experiments that were conducted in this chapter rely on manual evaluation of the test result and from the support of software metrics tool. As we know that some of the inputs are based on XMI files, we did not consider a faulty XMI generation by UML CASE tools. We also did not consider if the software metrics tool used was unable to extract some of the metrics from the XMI files.

4.8 Conclusion and Future Work

This chapter has provided an assessment of the reverse engineering capability of eight CASE tools (six commercial and two open source). We have assessed the tools by evaluating the reverse engineering features that are provided. In summary, all CASE tools are capable of performing reverse engineering from source code to class diagrams and package diagrams. Some of the tools can also reverse engineer sequence diagrams, but need a little help from the user. The tools also support various types of input formats other than source code, such as class or object files and executable files. Even

though these input formats offer additional options to the user, the resulting diagrams differ from the results from using source code as input.

Generally, there are not many differences between the capabilities of the CASE tools in reverse engineering into UML. Almost all the evaluated tools have relatively the same strengths and weaknesses: CASE tools do not completely show all class information, and CASE tools are also not capable of correctly and completely presenting the class relationships – especially aggregation and composition.

For future work, we propose this evaluation to be extended to larger systems to evaluate the scalability and performance of the tools. Also, future research in reverse engineering should try to come up with abstraction mechanisms for leaving out details and emphasize important information from RE-CD.

From this research, after the correctness and completeness of the RE-CD is identified, the studies on condensation of RE-CD in Chapter 7, 8 and 9 will consider the following information:

1. Aggregation and composition as association relationship.
2. Two directional relationships (with different direction) should present as bidirectional relationship.
3. Two directional relationships (with the same direction) should present as a single relationship.
4. Only several CASE tools are suitable to reverse engineer source code to the class diagrams.

Part II

UML Class Diagram Simplification

Chapter 5

Eliciting Developer's Views on Simplifying Class Diagrams

Class diagrams play an important role in software development. However, in some cases, these diagrams contain a lot of information that makes it hard for software maintainers to use them to understand a system. In order to reduce the amount of information in a class diagram, a method to simplify a class diagram is needed. This simplified class diagram can be obtained by leaving out details that are not needed by keeping the important information. In this chapter, we aim to discover how to simplify class diagrams in such way that the system is easier to understand. For this study, we performed a semi-structured survey to elicit the information about what type of information developers would include or exclude in order to simplify a class diagram. This study involved 32 software developers with 75 percent of the participants having more than five years of experience with class diagrams. As for the results, we found that the important elements in a class diagram are class relationship, meaningful class names and class properties. We also found that, in a simplified class diagram, GUI related information, private and protected operations, helper classes and library classes could be excluded. In this survey, we also tried to discover what types of features needed for automatic class diagram simplification tools.

This chapter is a more detailed version of a publication entitled **"Class Diagram Simplification: What is in the Developer's Mind?"**, In Proceedings of the 2nd International Workshop on Experience and Empirical Studies in Software Modelling (EESSMod 2012)

5.1 Introduction

UML models which are usually created during the design phase are often poorly kept up-to-date during the realization and maintenance phase[172]. As the implementation evolves, correspondence between design and implementation degrades from its initial design [118]. For legacy software, reliable designs are often no longer available, while those are considered valuable for maintaining such systems.

Normally, to understand a software system, a software developer needs both source code and design. When new software developers want to join a development group; they need a starting point in order to understand the whole project before they are able to modify. It is important for a software developer to have an overview of the system before they started the software maintenance activity. Tools that support during maintenance, re-engineering or re-architecting activities have become important to decrease the time software personnel spend on manual source code analysis and help to focus attention on important program understanding issues [22]. In Chapter 3, we have demonstrated that the forward design class diagrams constructed by the developers in the open source software development community typically consist of critical or key classes in the system. These classes are considered as the required relevant classes to understand the system. For this reason, this study specifically aims at simplifying UML class diagrams by leaving out unnecessary information without affecting the developer's understanding of the entire software. To this end, we have conducted a survey to gather information from software developers about what type of information should not be included in a class diagram and what type of information they focus on. We prepared a questionnaire that consists of 15 questions which were divided into 3 parts in order to discover this information. In total, there were 32 participants that are professional software developers in the Netherlands.

The chapter is structured as follows. Section 5.2 discusses related work and followed by Section 5.3 that describes the survey methodology. Section 5.4 describes the result and findings. We discuss our findings in Section 5.5 and present our conclusions in Section 5.6. This is followed by our suggestion for future work in Section 5.7.

5.2 Related Work

In this section, we discuss some studies that are related to the research in this chapter.

5.2.1 Eye Tracking

Yusuf et al. [183] performed a study on assessing the comprehension of UML class diagrams via eye tracking. They used eye-tracking equipment to collect a subject's activity data in a non-obtrusive way as the subjects are interacting with the class diagram in performing a given task. Also, audio and video were recorded on every

subject during these tasks. Their goal was to obtain an understanding of how human subjects use different types of information in UML class diagrams in performing their tasks. The subjects are asked to answer several questions by viewing the class diagrams. They created two types of questions: (1) questions that deal with the basics of the class diagram, and (2) questions related to the software design. They concluded that experts tend to use such things as stereotype information, colouring, and layout to facilitate more efficient exploration and navigation of class diagrams. Also, experts tend to navigate/explore from the center of the diagram to the edges, whereas novices tend to navigate/explore from top-to-bottom and left-to-right. Thus, subjects have a variation in the eye movements depending on their UML expertise and software-design ability in resolving a given task.

5.2.2 Software Visualization

Koschke [95] performed a study of software visualization (SV) usage domains: software maintenance, reverse engineering, and re-engineering. The goal of their study was to help to ascertain the current role of software visualization in software engineering from the perspective of researchers in these SV usage domains. Most of the questions were opinion-related questions, meaning that they asked the subject whether he/she thought that the visualization is appropriate (for example). Another part of the questionnaire asked what kind of instruments they use in visualizing software and how they visualize the software. The result of this study demonstrated that when the subject visualizing artifacts, only 13 out of 82 subjects answered using "UML". The most answered with "graph" (49 subjects). This survey suggested that a suitable way of visualization be achieved if we have a good understanding of when and why a certain visualization technique is used based on the user's purpose and their task. However, because of the variety of purposes and users' need, experiments in this field seem difficult.

Bassil et al. [21] conducted a study of SV tools that existed in the year 2000. This study focused on functionality, practicality, cognitive and code analysis aspects that users may be looking for at SV tools. The participants of this study were users of such tools in their industries or users that used SV in a research setting. The participants were asked to rate the usefulness and importance of these aspects in their SV tools, and came up with their own desires. In general, the participants were quite pleased with the SV tool at hand. Their study found that the reliability of SV tool was the most important aspect. This is followed by the ease of using the tool and the ease of visualizing large-scale software.

5.3 Survey Methodology

In this section, we describe i) How the questionnaire was designed and why; and ii) How the experiment was conducted.

5.3.1 Questionnaire Design

The questionnaire was organized into three parts, i.e. Part A, B and C. In total, there were 15 questions. In Part A, we aimed to discover information about the respondent's personal characteristics, knowledge and experience with UML class diagrams. Meanwhile, in Part B and C, we aimed to discover information about how the respondents indicate classes that should be included in a class diagram. For this survey, we divide this questionnaire into two different sets of questions. Both sets of questions had the same questions in Part A and C. However, we differentiated the questions in Part B. The questionnaire can be found at [129].

Part A: Personal Background

Part A consisted of six questions. Questions 1 to 4 in this questionnaire were intended to access the information about the status of the respondents, years of working with class diagrams, where they learned UML, and how the respondents rate their skills in class diagrams. In questions 5 and 6, we wanted to compare the respondents' preferences for software documents (i.e. UML models or source code) for understanding a system. This comparison is conducted to find if there is any influence of the respondents' answer based on their preferred software document.

Part B: Selected Cases

This part contained three questions and each question consisted of a class diagram. In this part, the respondents were required to mark information that can be left out in the provided class diagram without affecting their understanding of the system. They were also allowed to write any comments or suggestions according to what information they find unnecessary in a class diagram. The following systems were used in this survey:

1. **Automated Teller Machine (ATM) simulation system:** This fully functional ATM simulation system provides a class design and a complete implementation source code. The class design was made by using forward design. However, the class design only consists of class names and relationships. The complete software documents based on UML that were provided consists of 22 design classes. We reverse engineered[1] the system's source code to reconstruct the detail design of this system.
2. **Pacman Game:** Pacman's Perilous Predicament is a turn based implementation on the classic Pacman arcade game [44]. In this study, we utilized the diagram of the second phase (milestone) of this project since the number of classes in the class diagram is not too high. The amount of classes in the source code in this

[1]From the options of the list of CASE tool evaluated in Chapter 4, we use Enterprise Architect [153] version 7.5 to produce RE-CD.

Table 5.1: *Level of Detail Description*

No.	Class Diagram Elements	Low Level of Detail (LLoD)	Medium Level of Detail (MLoD)	High Level of Detail (HLoD)
1	Classes Names	YES	YES	YES
2	Attributes Names	NO	YES	YES
3	Type in Attributes	NO	NO	YES
4	Operations	NO	YES	YES
5	Operations Return Type	NO	YES	YES
6	Parameters in operation	NO	NO	YES
7	Relationships	YES	YES	YES

system is 17 while only 15 classes are stated in the class diagram design. Both forward and reverse engineered[1] designs were used in this survey.

3. **Library System:** Library System is a system that enables users to borrow books from the library. This system is taken from [55]. This complete system consists of 24 classes in the source code. The reverse engineered[1] design was used for this survey.

As mentioned, this survey consisted of two different sets of questions. This part differentiates the set of questions by providing different types of class diagrams. The information about the sets of class diagrams is shown in Table 5.1.

Level of detail (LoD) represents the amount of information that is used to specify models [117] (in this study: the UML class diagram). Different LoD was used to simulate different types of LoD that normally exist in a class diagram (as demonstrated in Chapter 3). In every set of the questionnaire, both Medium Level of Detail (MLoD) and High Level of Detail (HLoD) is used. Figure 5.1 shows how the class diagrams with different LoD are constructed and Table 5.2 briefly describes the types of class diagrams used in both sets of questionnaire.

In set A, ATM system in MLoD and Library System in HLoD were used and in set B, ATM system in HLoD and Library System in MLoD were used. In addition, we also used different sources of class diagrams by utilizing forward design and RE-CD to simulate the different flavors of class diagrams that exist in the software industry.

Part C: Class Diagram Indicators for Class Inclusion

This part consisted of six open-ended questions. The aim of these questions was to discover what developers think about which information is needed in a class diagram and which information could be left out. Table 5.3 describes the questions in part C.

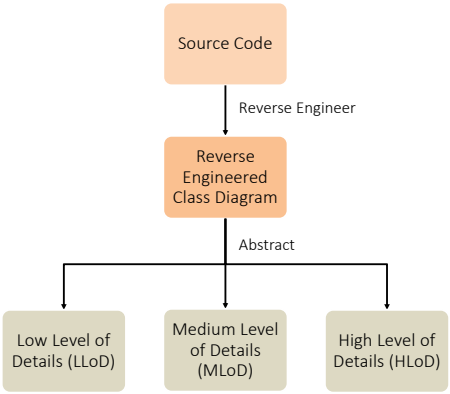


Figure 5.1: *Level of Detail Class Diagrams Preparation*

Table 5.2: *Information on Set A and Set B*

No.	Class Diagram	Set A	Set B
1	ATM System	Medium Level of Detail (MLoD)	High Level of Detail (HLoD)
2	Pacman Game	Forward Engineered Design	Reverse Engineered Design
3	Library System	High Level of Detail (HLoD)	Medium Level of Detail (MLoD)

5.3.2 Experiment Description

The experiment was conducted on the 6th of June 2012 at Leiden Institute of Advanced Computer Science (LIACS), Leiden. The participants of this survey were professional software developers from Devnology [102] (a software developer community from all over the Netherlands). In total, 32 Devnology members (out of 36) participated in the survey. This survey was conducted as part of the Devnology community "*Developer Back to School*" event at Leiden University. The participants had to answer every question and were free to ask any questions during the questionnaire session. The time given to answer the questionnaire was 60 minutes.

5.4 Results and Findings

This section presents our analysis and results of the answers given by the respondents. This section is divided into three parts. In the first part, we present our findings of part A (personal questions). The second part presents our findings of part B (selected cases). In the last part, we present our findings of part C of the questionnaire. The full responses to this survey are available at [131].

Table 5.3: *Detailed Explanation Part C*

No.	Question	Description
1.	**Question C1**: In software documentation, particularly in class diagrams, what type of information do you look for to understand a software system? (for example: relationships, operations, attributes, etc.)	To learn the type of information is important to understand the software system.
2.	**Question C2**: In a class diagram, what type of information do you think can be left out without affecting your understanding of a system? a. Classes (for example: Helper class, Interface class, Library class, …) b. Operations (for example: private, protected, public, constructor, …) c. Relationships (for example: labels, multiplicities, self-relations) d. Other(s):	To find out the type of information that can be left out from a class diagram.
3.	**Question C3**: Do you think that a class diagram should show the full hierarchy of inheritance? If not, which parts could be left out? (for example: parent, child, intermediate parent/child, leaf, …)	To find out what type of information on inheritance relationship is important.
4.	**Question C4**: What criteria do you think indicate that a class (in a class diagram) is important for understanding a system?	To discover the criteria developers use to decide a class is important in a class diagram.
5.	**Question C5**: If you try to understand a class diagram, which relationships do you look at first? (Example: dependencies, inheritance, associations, etc.)	To determine which relationship that can be considered important in a class diagram.
6.	**Question C6**: If there is a tool for simplifying class diagrams (e.g. obtained from reverse engineering), what features\functions would you expect from such a tool?	To find out what kind of features or functions are desired for a class diagram abstraction tool.

5.4.1 Part A: Personal Background

This part consists of six questions related to personal characteristics, knowledge and experience. We present our findings for each question as well as other related information.

Question A1: What is your role at the moment?

In this question, the respondents should state their role in software development. The choices of answers that have been given to the respondents are Project Manager, Architect, Designer, Programmer, and Tester. The respondents were allowed to select more than one answer. As for the results, 81% of the respondents are programmers and 50% of the respondents are software architects. As shown in Figure 5.2, 28% of the respondents are software designers. Figure 5.2 also highlights that the majority of the respondents are involved in the design and implementation phase in software development. This means, half of the programmers are involved in designing the software. All project managers that were involved in this study are also programmers. This indicates that all the respondents that participated in this study are directly involved in software development.

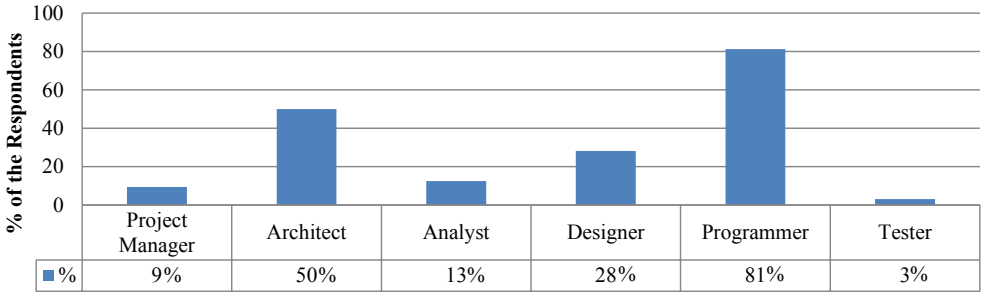


Figure 5.2: *Role of the Respondents*

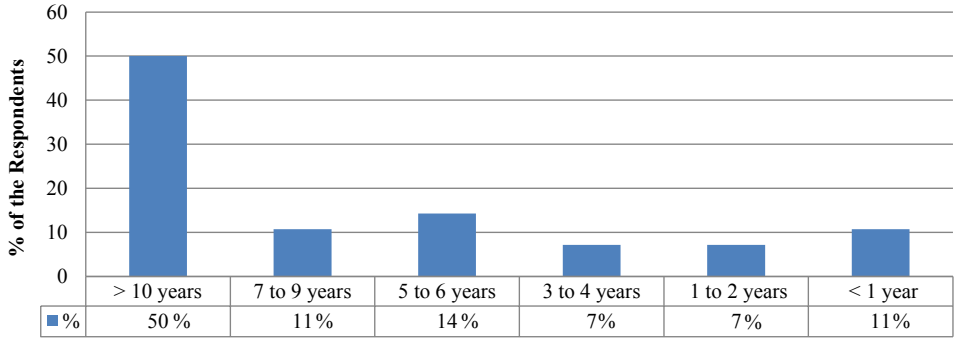


Figure 5.3: *Respondents Experience with Class Diagrams*

Question A2: How many year(s) of experience do you have in working with class diagrams?

Out of 32 respondents, 28 (88%) of the respondents answered this question. Figure 5.3 shows the complete results of this question. From these results, we found that 50% of the respondents are experienced with class diagrams for more than 10 years. The results also show that 75% of the respondents have experience with class diagrams for more than 5 years. Only about 11% (3 respondents) have less than 1 year experience in class diagrams. Even though they have less experience in class diagrams, they have the knowledge about UML as indicated by the answer of Question A3.

Question A3: Where did you learn about UML?

This question was intended to gather the information on (1) where the respondent learned about UML and, (2) whether all the respondents know about UML or not. The respondents were allowed to choose more than one answer. The choices were the following: Did not learn UML, From Colleagues/Industrial Practice, Professional Training, Learn by Myself, and polytechnic/University. The results show that 47% of

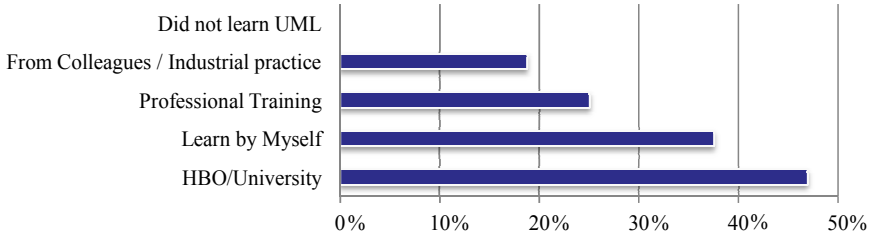


Figure 5.4: *Where did the Respondent Learn about UML*

the respondents had learned about UML in polytechnic or University and 25% have taken professional training to learn UML. This indicates that 72% of the respondents had formal training on UML. Meanwhile, 38% of the respondents learned UML by themselves and 19% learned from their colleague(s) or industrial practice. There were no participants that answered 'No'. This shows that all participants of this survey have some knowledge of UML. Figure 5.4 shows the complete results of this question.

Question A4: How do you rate your own skill in creating, modifying and understanding a class diagram?

This question was aimed to gain knowledge about the skills of the respondents in creating, modifying, and understanding class diagrams. Based on Figure 5.5, most of the respondents (88%) have average and good skills in creating, modifying, and understanding class diagrams and only 3% have excellent skills related to class diagrams. This indicates that over 90% of the respondents have average skills or above related to class diagrams. Meanwhile, 2 respondents (6%) have low skills and only 1 respondent (3%) has poor skills related to class diagrams. The 2 respondents that have low skills are software architects (with no other role) and the only one respondent that has poor skills is a programmer (with no other role).

Question A5: Indicate whether you (dis)like to look at the source code for understanding a system? + Question A6: Indicate whether you (dis)like to look at UML models for understanding a system?

Question A5 and A6 were aimed to discover the respondent's opinion about the usage of UML and source code as an artifact to understand a system. Most of the respondents of this survey are programmers and we expected that the respondents would choose the source code over UML. To present this result, we combined those two questions for a comparison between the respondent's like or dislike for UML and the respondent's

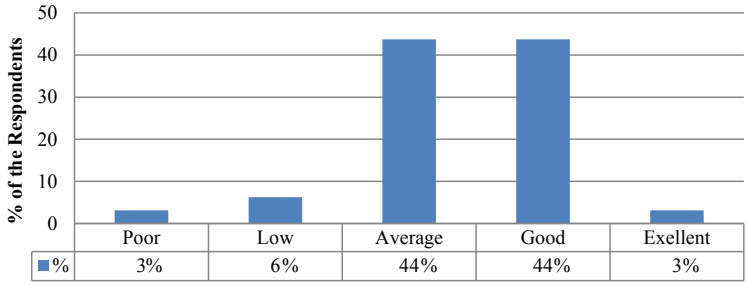


Figure 5.5: *Respondent's skill on Class Diagram*

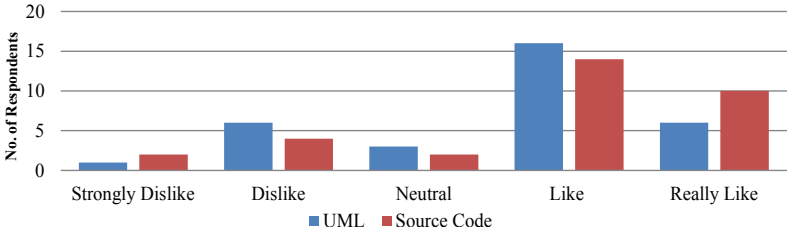


Figure 5.6: *Respondents Like or Dislike Source Code vs UML*

like or dislike for source code. The results shown in Figure 5.6 indicate that in general, there is no substantial difference between Like or Dislike of source code versus UML design. We may say based on this result that even experienced programmers found that UML is helpful for system understanding.

We further investigated this result by separating this according to the role of the respondents (specifically programmer, software architect, and software designer). Figure 5.7 shows the results of question A5 and A6 for respondents with the role of a programmer. The results show that the programmers are a bit more positive about source code than UML, but the difference is not substantial. These results seem almost the same as the overall results shown in Figure 5.6.

It was quite a surprise to see that a lot of software architects like using source code more than UML to understand a system (Figure 5.8). The same goes for the software designers; they like using source code more than UML to understand a system (Figure 5.9).

Others:

Combination of Question A1 & A4

The combination of results in question A1 and A4 is shown in Figure 5.10. This

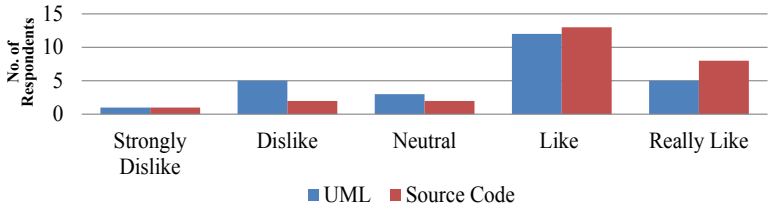


Figure 5.7: *Programmers Like or Dislike Source Code vs UML*

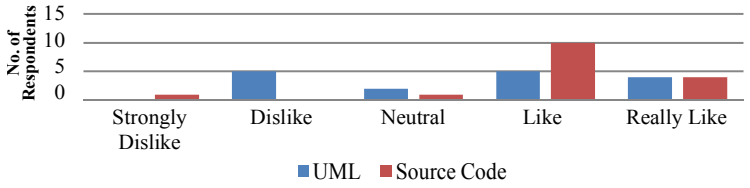


Figure 5.8: *Software Architects Like or Dislike Source Code vs UML*

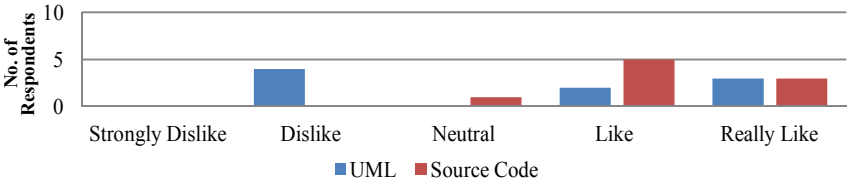
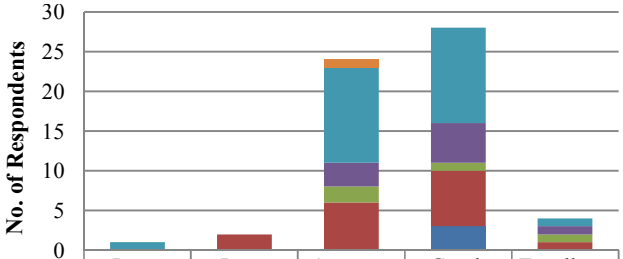


Figure 5.9: *Software Designers Like or Dislike Source Code vs UML*



	Poor	Low	Average	Good	Excellent
Tester	0	0	1	0	0
Programmer	1	0	12	12	1
Designer	0	0	3	5	1
Analyst	0	0	2	1	1
Architect	0	2	6	7	1
Project Manager	0	0	0	3	0

Figure 5.10: *Class Diagram Skill per Role*

result surprisingly shows that there were software architects that rated themselves poor in creating, modifying, and understanding class diagrams. However, based on our informal interview with these respondents, a software architect mentioned that they only use boxes and lines for their architectural work. This may be the reason why there are software architects that have a poor skill in class diagrams.

5.4.2 Part B: Selected Cases

In Part B, the respondents have been provided with three class diagrams from different systems and domains. Those class diagrams also varied in level of details (1 x LLoD, 1 x MLoD, 1 x HLoD). The results of this part were analyzed by combining the answers based on the following categories: Attribute, Operation, Class, Relationship and Package.

Category 1: Attribute

This category is divided into two subcategories: Properties and Types of Attribute. We divided the Properties subcategory into three elements: Protected, Public and Private. This basically means that if a respondent marked the private variables in a class diagram or suggested to exclude the private attribute, we assumed that the respondent chose not to include the private attribute element in class diagrams. We also divided the Types of Attribute subcategory into three elements: No primitive type, GUI related, and Constant. No primitive type means attribute that does not have any primitive type. GUI related attributes are attributes that relate to Graphical User Interface (GUI) libraries that are provided by development tools such as Textbox, Label and Button.

Figure 5.11 shows the results of the Attribute category. 25% of the respondents indicate a preference to leave out the GUI related attributes. For these respondents, this information seemed not important and based on our informal interview, the respondents were more concerned with classes that are created specifically for the application. 19% of the respondents prefer to leave out Private and Constant types of attributes. 13% of respondents propose to leave out Protected attributes. 3 out of 32 respondents (9%) think that all attributes should be left out. These respondents commented they only need class names and relationships in a class diagram.

Category 2 : Operation

The results of the Operation category are presented in Figure 5.12. The results show that 25% of the respondents chose to exclude the Constructors Without Parameters. This type of operation is not important because it does not convey any important information because the default initialization of an object is without parameters. Nevertheless, 16% of the respondents suggested that all Constructors could be left out

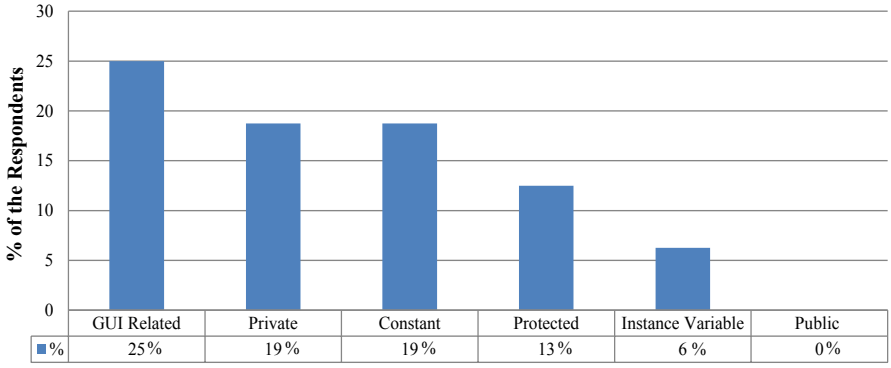


Figure 5.11: *Information of Attribute that Could be Left out*

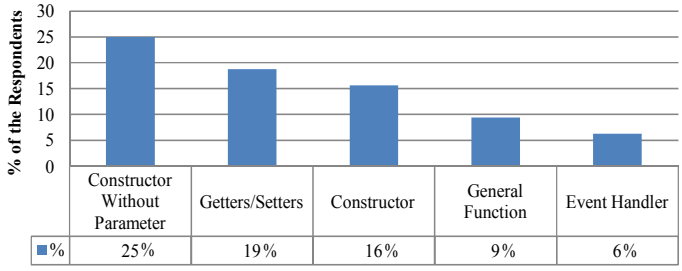


Figure 5.12: *Types of Operation that could be Excluded in Class Diagrams*

in a class diagram. For Getters and Setters, 19% of the respondents suggested that these operations should be excluded from class diagrams. The reason for this could be that it is a common operation that is created for accessing and modifying variables in a class diagram. 9% of the respondents mentioned that General Functions should not be included in class diagrams because these functions are commonly used and well-known to programmers. Apart from the result presented in Figure 5.12, 15% of the respondents indicated that all operations should be excluded from a class diagram. These respondents mentioned that only class names and relationships are needed in a class diagram.

Category 3: Class

Based on the respondents' answers, we divide the class category into two subcategories: (1) Types of Class and (2) Role (figure 5.13). The Types of Class subcategory consists of Interface, Enumeration, and Abstract elements while the Role subcategory consists of five elements which are Console, Listener, Input/Support Classes, Log, and GUI Related. The Role means that the class(es) have a specific role in the system.

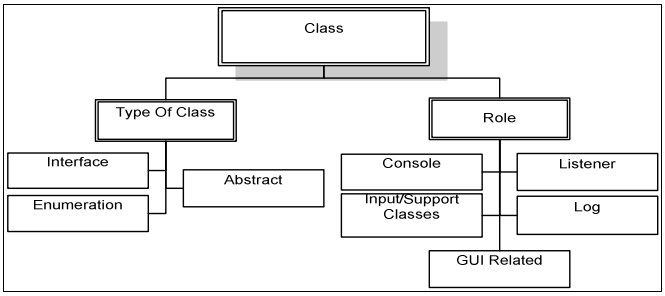


Figure 5.13: *Class Category*

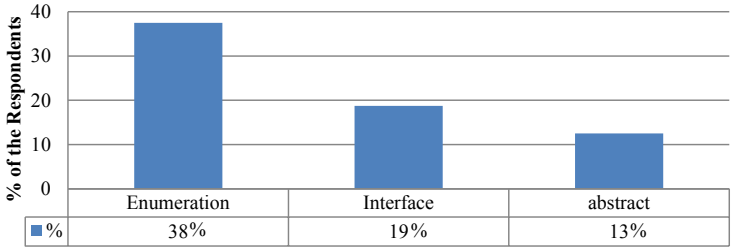


Figure 5.14: *Types of Class that could not be Included in Class Diagrams*

For the subcategory Types of Class (Figure 5.14), 38% of the respondents chose not to include Enumeration classes. Enumeration classes are classes whose values are enumerated in the model as enumeration literals, which are not needed to understand a system. This is followed by Interface classes with 19% and 13% suggested that Abstract classes should not be included in simplified class diagrams.

Figure 5.15 shows the Role subcategory results. It shows that half of the respondents suggested that GUI related classes and classes for logging tasks could be left out in order to simplify a class diagram. Most GUI related classes are present in the Library system and the Log class exists in the ATM system. The respondents suggested eliminating these classes because without these classes they can still understand the system. The Input role refers to classes that are used to take the input from the interface that directly interact with the actor of the system. In the case of the ATM system, the "Money" and "Card" classes are an example of input classes. 22% of the respondents said that this type of class could be omitted from a class diagram. The "Console" and "Listener" functions appear in the Pacman Game. These classes can be considered as classes that interact with the user input and the other system input. 6% of the respondents chose to exclude the listener classes from the class diagram while 3% of the respondents chose to exclude the console classes.

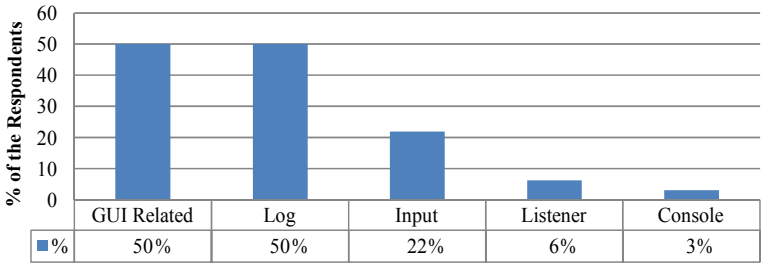


Figure 5.15: *Class Role that could be Excluded in Class Diagrams*

Category 4: Relationship

The Relationship category is divided into two subcategories, which are Relationship Label and Coupling ≤ 1. Almost all the respondents that participated in this survey agreed that the Relationship element is important in a class diagram. However, there are some information related to the Relationship element that could be omitted from a class diagram. 31% of the respondents intend to exclude classes with Coupling ≤ 1 because it seems that classes that only have coupling ≤ 1 are not important and more seen as a helper class. 6% of the respondents chose to remove the relationship labels.

Category 5: Package

The package category is introduced because there were several respondents that separated the class diagram in such way that there were two or more class diagrams instead of one. The amounts of classes in the three class diagrams range from 15 to 22. Specifically, in the Library System class diagram, there were 4 respondents that drew several lines to separate the GUI related classes from the classes that were created by the developer. They suggested that the class diagram should be separated into two different diagrams. This basically means that they wanted to keep the GUI related classes and classes created for the application separately.

5.4.3 Part C: Class Diagram Indicators for Class Inclusion/Exclusion

Part C consists of six open-ended questions. The analysis of this part was done by observing the answers from the respondents and creating several keywords to categorize these answers.

Type of Information in Class Diagrams for Understanding a Software System

The respondents were asked the following question: *'In software documentation, particularly in class diagrams, what type of information do you look for to understand a software*

Table 5.4: *Keywords on Types of Information to Understand a System*

No	Category	Keywords	No	Category	Keywords
1	**Relationship / Connectivity / Interaction**	*Association*	3	**Class structure / properties**	*Abstraction*
		Inheritance			*Method/Operation*
		Direction			*Attribute*
		Dependency			*Public Interface*
		Multiplicity			*Class Entities*
2	**Class Semantic**	*Classname (meaningful)*			*Size Large/Small*
		Class Behaviour			*Public Properties*
		Business Entities			*Class Hierarchy*
		Main Classes/Object/Purpose			*Object related*
		Class functionality and responsibility	4	**High level**	*Concept*
		Domain			*Design Pattern*
		Properties name and methods name			*Overview*
		Reasoning	5	**Others**	*Data*
		"Starting" point			*All Generic Classes*
		Optional info			

Figure 5.16: *Types of Information the Respondents Look for in Class Diagrams*

system?'. Based on the answers, we created several keywords and categories as shown in Table 5.4. The results in Figure 5.16 shows that class relationships are the most important information in a class diagram that the respondents searched for, in order to understand a class diagram. 81% of the respondents mentioned this. 59% of the respondents searched for class Role and Responsibility (RnR) such as meaningful class names and class functionality and behaviour. 44% of the respondents were looking at class properties such as attributes, operations and class interfaces. This is followed by 34% of the respondents that were looking at the high-level abstraction of the class diagram for example design concepts, design patterns and class overviews.

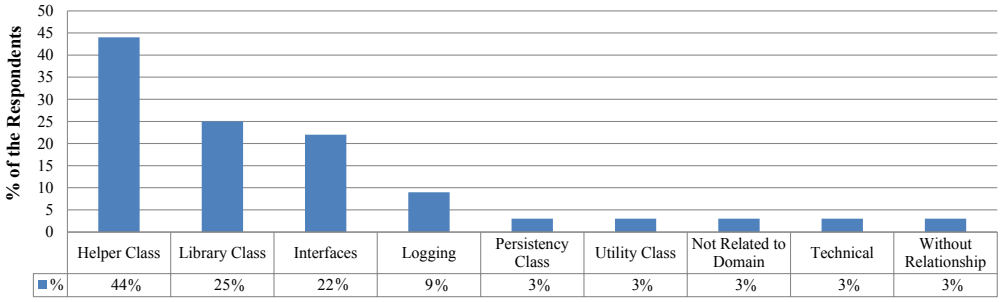


Figure 5.17: *Information of Classes that could be Omitted*

These results show that the relationships between classes are important in class diagrams. It is the primary information in a class diagram that most of the software developers look into. The semantics of a class such as a meaningful name of classes, operations and attributes also play a major role to assist the software developer in understanding a system.

Type of Information that could be Omitted/Excluded

We asked the respondents to answer the following question: *'In a class diagram, what type of information do you think can be left out without affecting your understanding of a system?'*. The results are divided into four sections, which are: Classes, Operations, Relationships, and Others.

In the section of classes, almost half of the respondents (44%) suggested that helper classes could be omitted from a class diagram (see Figure 5.17). A quarter of the respondents (25%) did not want library classes to appear in a class diagram. These library classes could make a class diagram more complex. 22% of the respondents suggested that the interface classes could be omitted from a class diagram.

In the section of Operations, the results (Figure 5.18) show that 66% of the respondents chose to exclude private operations in a class diagram. 56% of the respondents mentioned that constructors and destructors are not needed in a class diagram in order to understand a system while only 9% of the respondents mentioned that they do not need constructors without parameters. 41% of the respondents mentioned that protected operations could be left out from a class diagram. A reason for this could be that this type of operation can be assumed as a private operation, but appears public to several classes only. It was quite a surprise that not many respondents suggested removing getters and setters from the class diagram since these operations can be integrated into other operations that a system actually needs.

In section Relationship, multiplicity is the most respondents mentioned not needed in a class diagram. However, only 6% of the respondents mentioned this, which is a

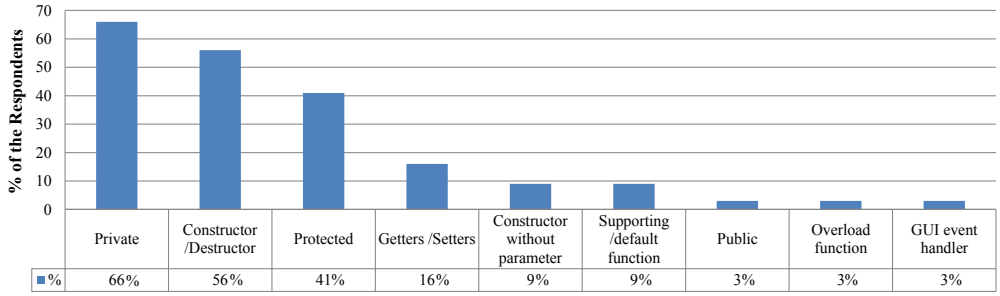


Figure 5.18: *Information of Operations that could be Omitted*

quite low percentage. 3% of the respondents do not need any Labels (or roles of the relationships), Self Relations and References in a class diagram.

In terms of "Other information" in class diagrams that could be omitted, 9% of the respondents stated that the private fields could be omitted from a class diagram. Only 3% of the respondents suggested not to include technical, duplicates and UI information in class diagrams.

The Criteria to Indicate Important Classes in Class Diagrams

We asked the respondents *'What criteria do you think indicate that a class (in a class diagram) is important for understanding a system?'*. This question aimed to discover the criteria to indicate important classes in a class diagram. As shown in Figure 5.19, 38% of respondents think that the relationships are the most important criterion in a class diagram. This also aligns with the results for question C1. 16% of the respondents mentioned the following criteria are important in a class diagram: Meaningful class name; Business or domain value; and Class position. Several respondents prefer to search for the position of the class (in the layout) and most of the respondent's mentioned that classes located in the middle of a class diagram are the important classes.

Type of Relationships that the Respondents Look at First

We asked the respondents *'If you try to understand a class diagram, which relationships do you look at first?'*. In this question, we aimed to find out what type of relationship the respondents look at first and three types of relationships were provided as example answers (composition, aggregation, and realization). The results in Figure 5.20 show that 41% of the respondents looked for association relationships, 19% searched for dependency relationships, and 9% searched for inheritance relationships.

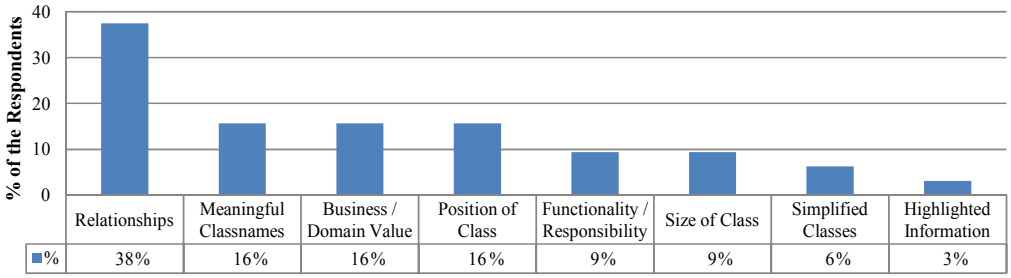


Figure 5.19: *Important Criteria in a Class Diagram for Understanding a System*

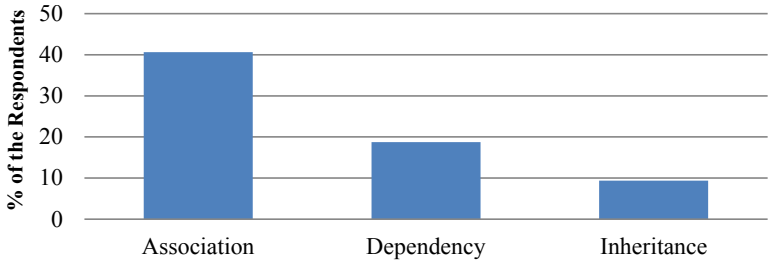


Figure 5.20: *The Type of Relationship in Class Diagrams that the Respondents Look at First*

Features/functions Expected in a Class Diagram Simplification Tool

We asked the respondents *'If there is a tool for simplifying class diagrams (e.g. obtained from reverse engineering), what features/functions would you expect from such a tool?'*. In this question, we tried to discover what kind of features the respondents are looking for if there is a tool that could simplify a class diagram. The results (see Figure 5.21) show that the respondents mainly want a tool that can hide/unhide information. The other feature that relates to this is the drill up/down feature. 16% of the respondents wanted to see more information about a class by hovering over a class in a class diagram for example. Another feature that many respondents (13% of the respondents) wanted is the changeable layout of the class diagram in which the navigation can be improved.

5.5 Discussion

In this section, we discuss the results and findings presented in the previous section. The discussion is divided into four subsections: Class Properties, Class Role and Semantics, Class Diagram Simplification Tool Features, and Threat to Validity.

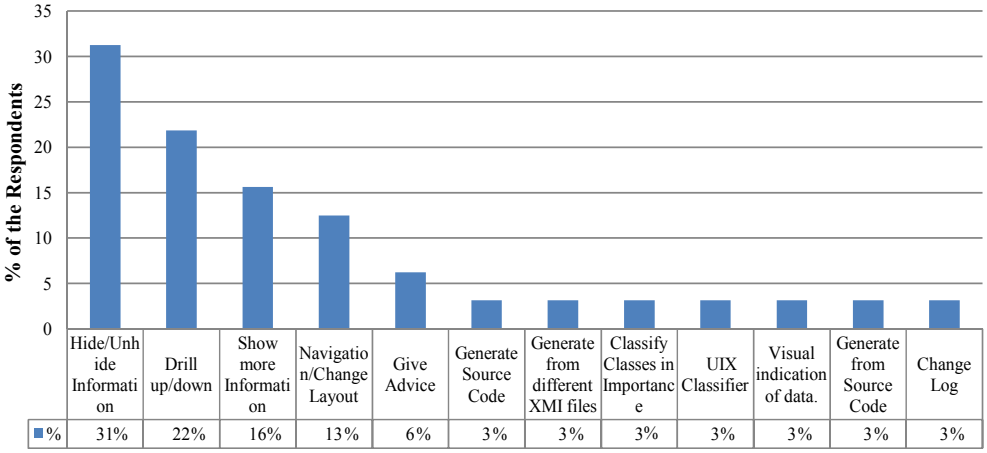


Figure 5.21: *The Features that a Tool Should have for Simplifying UML Class Diagrams*

5.5.1 Class Properties

Relationships in a class diagram are considered the important elements to understand a system through class diagrams. Most of the respondents in this survey looked at the association relationship first. This shows that the association relationship is important in class diagrams. In this survey, we found that most of the respondents suggested leaving out or separating the GUI related information from the class diagrams. The respondents focus more on application-specific class diagrams information. The GUI classes appear in RE-CD because Rapid Application Development (RAD) tools inject or generate source code for this. In terms of class operations, most of the respondents suggested leaving out the private and protected type of operations. These types of operations are only used for internal classes and member classes for protected operations. We also discovered that constructor/destructor operations could be omitted from class diagrams. Particularly in Part B, we found that most of the respondents suggested that constructors without parameters should be excluded.

5.5.2 Class Role and Responsibility (RnR)

One of the useful discoveries in this study is the importance of the class RnR in a class diagram. By using this information, they can get an overall idea of how the system works and get some hints of the functionalities of classes in a class diagram. In this survey, we also discovered that classes that could be left out in a class diagram (in the context of system overview) are helper classes, library classes and interface classes. Most of the respondents suggested leaving out helper classes. Nevertheless, it is not easy to automatically identify helper classes.

5.5.3 Class Diagram Simplification Tool Features

This study found that the desired features for a simplification tool are to hide/unhide information; and drill up and down a class diagram. These features are desired because they help to zoom in and zoom out in class diagrams. From our informal discussion with the respondents, simplification of class diagrams is needed when they want to understand the overall system design, but detailed information in class diagrams is needed for modification tasks. Hence, both simplified and detailed diagrams are essential.

5.5.4 Threats to Validity

In this subsection, we discuss the internal and external threats to the validity of this survey.

Internal validity: The three selected cases used in Part B are considered as medium size. The result may be different if a larger size of software system is used. In this survey, we concerned about the time constraint for the respondents to complete the survey. By using these selected cases, the survey was able to complete in given time frame. Also, we believe that the class diagrams in these selected cases are representative.
External validity: We acknowledge that the number of respondents in this is small. However, we showed that 75% of these respondents have experience more than 5 years in class diagrams. We also believe that a survey and informal discussion about class diagrams with 32 professional developers in the same place and time contribute to a significant result.

5.6 Conclusion

This chapter presented a study on how to simplify a class diagram without affecting their understanding of a system. In particular, the questions in this survey were about what information could be left out from a class diagram and also what kind of important information should remain. 32 professional software developers from the Netherlands participated in this survey.

From the results, the most important elements in class diagrams are the relationships. Class relationships are important to show the structure of a system. The type of relationships that the developers look at first are the association and dependency relations. In this survey, we discovered that the class diagram's role and responsibility are important because most of the respondents search for meaningful class names and class roles in order to get a high-level understanding of how a system works. This means, meaningful class names, operation names and attribute names are important for system understanding.

To simplify a class diagram, most of the respondents chose to exclude GUI related information and also library classes. This shows that most of the software developers focus on application-specific classes, but not the generic or utility classes. Most of the respondents also mentioned that helper classes should be excluded to simplify a diagram. However, it is not easy to automatically identify helper classes. Private operations, protected operations and constructors (without parameter) are types of operations that could be left out in order to simplify a class diagram. These types of operations seem not to be important. Although we are aware that research on validation of our approach needs to be done, we found several useful indicators that could be used for class diagrams simplification.

5.7 Future Work

This study was an early experiment on how to simplify class diagrams and we see a number of ways to extend this work. In Part B, we have used RE-CDs and forward engineered class diagrams in two separate groups. Also, we have used different Levels of Detail in different sets of groups. A comparison of these different flavors of class diagrams in terms of what information the respondents suggest to leave out can be the future work to extend this study. It would be interesting to compare the results between these class diagrams and see if there are any differences in what the software developers are excluding from these diagrams. We propose to validate the resulting class diagram by using an industrial case and discover the suitability of the simplified class diagram for the practical usage.

From the results, we found that class role and responsibility are one of the important indicators in a class diagram. The role and responsibility of a class are detected by using the class names, operations names and attributes names. We suggest a study on the names (classes, operations and attributes) that the software developers find important or meaningful in order to understand a system. The results of this study are used to predict the important classes in a class diagram in chapter 8.

Chapter 6

Exploring the Suitability of Object-Oriented Design Metrics as Features for Class Diagram Simplification

Class diagrams may include an overwhelming amount of information. For large and complex class diagrams, there is a possibility that not all information in the class diagram is important for understanding the system. In this chapter, we study how to identify essential and secondary information in class diagrams. To this end, we performed a survey with professionals, academics and students to enquire how to decide which information in class diagrams is considered important. In particular, we explore whether software design metrics can be used as a means of identifying important classes in a class diagram.

In total, 25 complete responses were received. 76% of the respondents have average or above skills with class diagrams. We discovered that the metric that counts the number of public operations is the most important metric for indicating the importance of a class in a diagram. Also, we discovered that class names and coupling were influencing factors when it comes to excluding classes from a class diagram.

This chapter is a more detailed version of a publication entitled **"UML Class Diagram Simplification - A Survey for Improving Reverse Engineered Class Diagram Comprehension"**,In Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development(MODELSWARD 2013)

6.1 Introduction

The UML class diagram is one of the valuable artifacts in software development and software maintenance. This diagram is helpful for software developers and software maintainers in order to understand architecture, design, implementation and behavior of software systems. UML class diagrams describe the static structure of programs at a higher level of abstraction than source code [70].

Reverse engineering is one of the possible techniques to discover a software design after the implementation phase. Reverse engineering is the process of analyzing the source code of a system to identify its components and their interrelationships and create design representations of the system at a higher level of abstraction [41]. With this technique, recovery of a class diagram can be done based on the source code. However, the resulting class diagrams from reverse engineering techniques often suffers from too much detail and information. In particular, RE-CDs are typically a detailed representation of the underlying source code, which makes it hard for the software engineer to understand what the key elements in the software structure are [126]. Although several Computer Aided Software Engineering (CASE) tools have options to leave out several properties in a class diagram, they are unable to automatically identify classes and information that are not useful or less important. As part of a recent study [59], Fernández-Sáez et al. found that developers experience more difficulties in finding information in reverse engineered diagrams than in forward designed diagrams and also find the level of detail in forward designed diagrams more appropriate than in reverse engineered diagrams. For this reason, the information that is needed by developers or maintainers to be shown in a class diagram should be discovered.

In this chapter, we aim at simplifying UML class diagrams by leaving out unnecessary information without affecting the developer's understanding of the system. Based on the feedback in Chapter 5, the software developers indicated that the system structural information (e.g. relationship and class elements) influence the determination of classes that could be included (inclusion) and classes that could be omitted (exclusion) in the class diagrams. To this end, we have conducted a survey to gather information from IT professionals, researchers or academics and students about what type of information they focus on. The survey directed to the structural information based on the object-oriented design metrics (i.e. the metrics from size category, inheritance category and coupling category). We prepared a questionnaire that consisted of 24 questions that are divided into 3 parts in order to discover this information.

The chapter is structured as follows. Section 6.2 discusses related work. Section 6.3 describes the properties and tools for this research. Section 6.4 explains about the survey methodology while Section 6.5 presents the results and findings. We discuss our findings in Section 6.6. This is followed by our conclusions in Section 6.7 and future work in Section 6.8.

6.2 Related Work

In this section, we discuss several studies that are related to the research in this chapter.

6.2.1 Usage of design metrics

Design metrics have been used for various purposes in software engineering. The Chidamber and Kemerer metrics (widely known as CK metrics) are well-known object-oriented metrics that is commonly used in software maintenance and fault proneness research.

In software maintenance, Li et al. [104] use object-oriented metrics (CK) to predict software maintenance effort. Their study shows a strong relationship between maintenance effort and object-oriented metrics. Their study also validates the results to prove the ability of these metrics to predict the maintenance effort. Binkley et al. [27] investigate more specific to coupling metrics in predicting maintenance measure. Their study found that the coupling dependency metrics (CDM) is suitable for predicting maintenance measures (e.g. a low level of coupling undergoes less maintenance and have fewer maintenance faults). They also found that CDM also suitable to predict the number of run-time failures.

There are several works on the usage of object-oriented design metrics associated with fault-proneness. Briand et al. [36] [35] found that several CK metrics were associated with the fault-proneness of classes. Also, Tang et al. [163] found that the Weighted Methods Per Class (WMC) and Response for a Class (RFC) metrics are associated with fault-proneness. El Emam et al. [53] found that inheritance and Export Coupling (EC) Metrics are associated with fault-proneness. In later research, Gyímothy et al. [73] discovered the Coupling between object (CBO) is the best metrics to predict fault-proneness.

In our study, we used several object-oriented metrics to predict the class should be included and should be excluded in the class diagram. We measure the influence of the object-oriented metrics in predicting class inclusion/exclusion by the score ranking based on the respondents' answers.

6.2.2 Automated Abstraction of Class Models

Falleri et al. [56] proposed an approach for class model normalization to produce a simplified class diagram by removing redundant information. The Relational concept analysis and Formal Concept analysis are used to process the normalization of the class model. Similar to our study, they also suggest that the usage of elements name for class model abstraction.

Egyed [51][52] proposed an approach for automated abstraction that allows designers to "zoom out" on class diagrams to investigate and reason about their bigger picture. This approach was based on a large number of abstraction rules. In total, the

article provides 121 rules to abstract a class diagram. However, this work is more concentrated on the abstraction of classes relationship. It could not automatically select the classes that should be included and should be excluded in class diagrams.

In our study, we conduct a survey to find out which object-oriented design metrics influences the software developer in selecting the classes that should be included and excluded.

6.3 Examined Properties and Tools

In this section, we describe i) the design metrics that we consider, and ii) the tools used for this research.

6.3.1 Examined Properties

SDMetrics [180] is an object-oriented design measurement tool for the Unified Modeling Language (UML). SDMetrics is capable of measuring 32 types of class diagram metrics which are divided into five categories, namely Size, Coupling, Inheritance, Complexity and Diagram. However, in this study, we only used 14 metrics from the categories of Size, Coupling and Inheritance. These 14 metrics are selected based on our initial experiments in Chapter 3. We reverse engineered the source codes of the case studies presented in Chapter 3 and extracted all object-oriented design metrics provided by SDMetrics. We found that only 14 metrics (as shown in Table 6.1) have significant value for measurement.

6.3.2 Tools

SDMetrics [180] is used to measure the structural properties of object oriented design. SDMetrics version 2.11 (academic license) is used for this purpose. We chose Enterprise Architect [153] version 7.5 for creating forward design and RE-CDs for this survey.

6.4 Survey Methodology

This subsection describes the design of the questionnaire. We explain how the questionnaire was designed and why. We also describe our online survey experiment that explains how the experiment was conducted.

6.4.1 Questionnaire Design

The questionnaire consisted of 3 parts i.e. part A, B and C. There was a total of 24 questions in this questionnaire.

Table 6.1: *The Chosen Software Design Metrics [180]*

No.	Metrics	Category	Description
1	NumAttr	Size	The number of attributes in the class.
2	NumOps	Size	The number of operations in the class.
3	NumPubOps	Size	The number of public operations in a class.
4	Setters	Size	The number of operations in a class with a name starting with 'set'.
5	Getters	Size	The number of operations in a class with a name starting with 'get', 'is', or 'has'.
6	NOC	Inheritance	Number of Children (NOC) calculates the number of immediate subclasses subordinated to a class in the class hierarchy.
7	DIT	Inheritance	Depth Inheritance Tree (DIT) calculates the longest path from the class (in the class diagram) to the root of the inheritance tree.
8	CLD	Inheritance	Class Leaf Depth (CLD) calculates the longest path from the class to a leaf node in the inheritance hierarchy below the class.
9	Dep_Out	Coupling (import)	The number of dependencies where the class is the client.
10	Dep_In	coupling (export)	The number of dependencies where the class is the supplier.
11	EC_Attr	Coupling (import)	The number of times the class is externally used as attribute type.
12	IC_Attr	coupling (export)	The number of attributes in the class have another class or interface as their type.
13	EC_Par	Coupling (import)	The number of times the class is externally used as parameter type.
14	IC_Par	coupling (export)	The number of parameters in the class have another class or interface as their type.

In part A, we aimed to discover the respondent's personal characteristics and experience with class diagrams. Meanwhile, Part B aimed to discover what object-oriented design metrics that the respondents find influential in considering classes that could be included in class diagrams. In part C, we aimed to discover what classes the respondents leave out when looking at a diagram and what class diagram(s) the respondents prefer when looking at different types of class diagram designs. This is an online questionnaire and is hosted by LimeSurvey [5] and a printable version is available at [130].

Table 6.2: *Answers Multiple Choices Questions*

Choices	Answers
A	Class(es) **definitely should not** be included
B	Class(es) **probably should not** be included
C	Class(es) **sometimes** be included
D	Class(es) **probably should** be included
E	Class(es) **definitely should** be included

Part A: Background of the Respondents

Part A consisted of 4 questions. Question 1 asked about the current status of the respondents. Question 2 intended to collect information about the respondent's location (optional question). We asked how many years of experience the respondent has with class diagrams in question 3. The last question asked the respondents to rate their skills in creating, modifying and understanding class diagrams.

Part B: Class Diagram Indicators for Class Inclusion /Exclusion

This part consisted of 14 questions. The first 13 questions asked about the influence of class diagram elements (based on object-oriented design metrics) to distinguish classes that should be included or excluded. In detail, we asked opinions of software developers on to whether they believe a particular metrics should be used for deciding whether a class should be included or excluded. In each question, we briefly explained about the metrics that was used in the question and five answers were offered. The choices of answers are shown in Table 6.2.

The last question of part B (i.e. question 14) is to discover the reason of the respondents for including and excluding a class in a class diagram. This question aimed to collect the information complementary to object-oriented design metrics about the reason of the respondents for including and excluding a class in a class diagram.

Part C: Practical Simplification Problems

Part C contained 6 questions. In this part, these well-known domain systems were selected to avoid bias about the domain knowledge of the respondents. The following class diagrams were involved in this survey:

1. **Automated Teller Machine (ATM) simulation system:** We used the forward design of an ATM simulation system [28] that only contains class names and class relationships. In total, there are 22 classes in this class diagram.
2. **Library System:** The Library System is a system that enables a user to borrow a book from a library. This system which was taken from [55] contains 24 classes. The RE-CD of this system was used in this questionnaire.

Table 6.3: *Description of the Class Diagrams Used in the Questions*

Question	System	Source of Diagram	Level of Detail (LoD)
C1	ATM Machine	Forward Design	Low
C2	Library System	Reverse Engineered	High
C3	Pacman Game	Forward Design	High
C5	Pacman Game	Reverse Engineered	High

Table 6.4: *Choices of Answers for Question 4*

Choices	Descriptions
A	I prefer class diagram A (ATM System)
B	I prefer class diagram B (Library system)
C	I prefer class diagram C (Forward design Pacman)
D	I prefer them all
E	I do not prefer them
F	It does not matter which one

3. **Pacman Game:** Pacman's Perilous Predicament is a turn-based implementation of the classic Pacman game. To accommodate its turn-based nature, the game play mechanics will be changed into more of a puzzle game. This project can be found at [44]. In this questionnaire, we used the diagram of the second phase (Milestone 2). We used two types of diagrams from this system, namely the forward design and the RE-CD. The forward design consists of 17 classes while the RE-CD contains 15 classes. The forward design was detailed and hence, similar to the source code.

We also tried to simulate the various flavours of class diagrams from the software industry by providing different Levels of Detail (LoD) of class diagrams and the sources of class diagrams. Different flavours of class diagrams allowed us to differentiate the indicators of class exclusion. The information about the class diagrams used in part C is shown in Table 6.3. Next to these 4 questions, we made another 2 questions in which we asked the respondent which class diagram he/she prefers. In the first question (question 4 in part C), the respondents were required to choose between an ATM system, a Library system and the forward design of a Pacman system. The respondents were also required to provide the reason they chose the answer. In the second question (question 6 in part C), the respondents were required to choose between the forward design and the RE-CD of Pacman. The respondents were also required to give the reason they chose the answer. Table 6.4 and 6.5 show the answer options for the multi-choice questions.

Table 6.5: *Choices of Answers for Question 6*

Choices	Descriptions
A	I prefer class diagram C (Forward design Pacman)
B	I prefer class diagram D (Reverse engineered design Pacman)
C	I prefer them Both
D	I do not prefer them
E	It does not matter which one

Table 6.6: *Total of Reponses*

Responses	Amount
Complete Responses	25
Incomplete Responses	73
Total Responses	98

6.4.2 Experiment Description

The experiment was conducted online (hosted by Limeservice [5]). The questionnaire was published online from 15th of May until the 3rd of August 2012.

We first invited students and researchers at the Leiden Institute of Advanced Computer Science (LIACS), Leiden, to our online questionnaire. Then, we promoted the questionnaire by using social media like Facebook, Twitter and LinkedIn. We also promoted this questionnaire to multiple online software developer forums.

The respondents were provided the facility to save the answers and the respondents could continue for a later time. The total respondents that entered this questionnaire were 98 (see Table 6.6). However, only 25 respondents completed this questionnaire. Most of the incomplete responses stopped after the questions in Part A.

6.5 Results and Findings

In this section, we present our results and findings from this survey. This section is divided into three subsections: Background of the Respondents, Indicator for Class Inclusion and Practical Simplification Problems.

6.5.1 Background of the Respondents (Part A)

This subsection presents the results of part A of the questionnaire in which we asked after the respondent's background information.

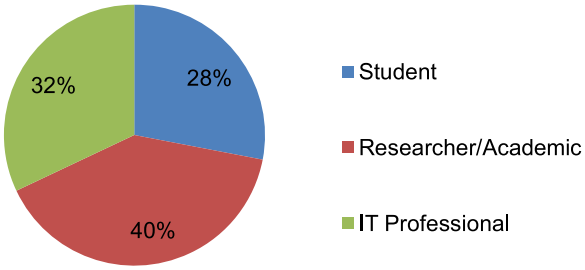


Figure 6.1: *Role of the Respondents*

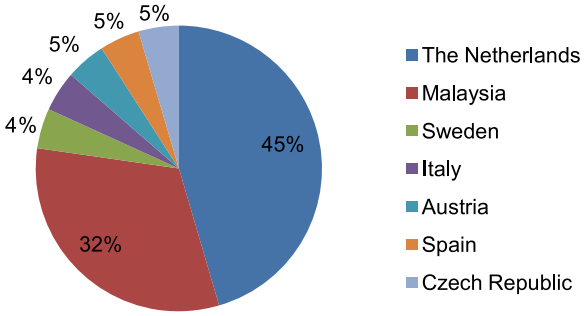


Figure 6.2: *Location of the Respondents*

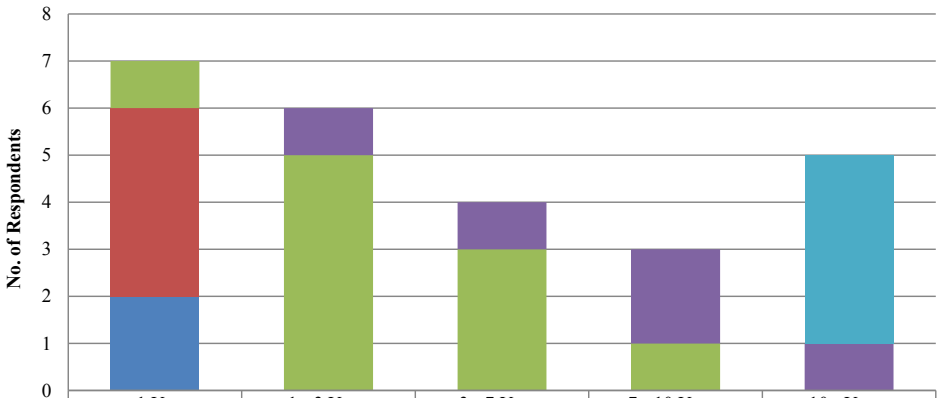
Roles and Locations

For the respondents' role, 40% of the respondents mentioned that their current status is Researcher/Academic while 32% of the respondents are IT Professionals. 28% of the respondents answered Student. None of the respondents answered "Other". Figure 6.1 shows the results of all the respondents. This result shows that the distribution of the respondent's status is quite even.

For the respondents' location, 45% of the respondents were from the Netherlands and 32% of the respondents were from Malaysia. The detail of respondents' location is shown in Figure 6.2.

Skills and Experience with Class Diagrams

For the years of experience in using class diagram, 28% of the respondents stated that their experience with class diagrams is less than 1 year. 24% of the respondents mentioned that their experience with class diagrams is between 1 and 3 years while 16% of the respondents answered this question with "3 - 7 Years". 12% of the respondents answered "7 - 10 Years" and 20% of the respondents mentioned that they had more than 10 years of experience with class diagrams.



	< 1 Year	1 - 3 Years	3 - 7 Years	7 - 10 Years	10+ Years
Excellent	0	0	0	0	4
Good	0	1	1	2	1
Average	1	5	3	1	0
Low	4	0	0	0	0
Poor	2	0	0	0	0

Figure 6.3: *Class Diagram Skill and Years of Experience*

In Question A4, we asked the respondent to rate his/her skills in creating, modifying and understanding class diagrams. 40% of the respondents answered Average, while 20% answered "Good". 16% of the respondents answered Excellent. 16% of the respondents rated their skill are Low and 8% of the respondents have Poor skill of class diagrams. This indicate that 76% of the respondents rated their skill of average or above. The complete results of the combination of these two questions are shown in Figure 6.3.

6.5.2 Indicator for Class Inclusion

In this subsection, we present the result of part B. This part consisted of 14 questions. Each of these questions asked whether some metric-value could be used as an indicator of the importance of a class. For example,

" **B3**: *Do you think that a high number of Public operations (NumPubOps) is an indicator that a class should be included in a class diagram? "*

For questions B1 to B13, the respondents were provided 5 answer options (see Table 6.2) to be chosen as their answer. We analyzed these 13 questions by using a score-system. The score-system is shown in Table 6.7.

The metrics are grouped into three categories which are: Size, Coupling and Inheritance. The details of these metrics are explained in Table 6.1. Figure 6.4 shows the results of the Size category. The average score for the metrics in category of size is 18.3.

Table 6.7: *Score-System Metrics - Question B1-B13*

Answer	Score
Class(es) **definitely should not** be included	-2
Class(es) **probably should not** be included	-1
Class(es) **sometimes** be included	0
Class(es) **probably should** be included	1
Class(es) **definitely should** be included	2

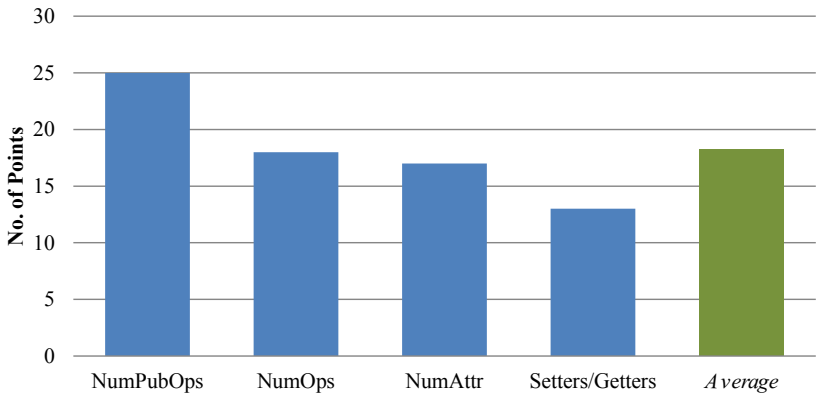


Figure 6.4: *Score Size Category (Question B1-B4)*

From these results, we found that operations are very important in class diagrams. In particular, public operations. This finding aligns with our findings in Chapter 5 where the respondents did not like to see private and protected operations. In other words, they find public operations better indicators for inclusion in a class diagram. As for setters/getters, these have the lowest score in this category. This indicates that the setters/getters are not an important element in a class diagram for the respondents. A reason for this could be that it is a common operation. NumAttr and NumOps also have an average amount of points. We can say that these metrics are normally needed in a class diagram, but public operations are more preferred.

Figure 6.5 shows the results of the Coupling category. On average, the score for the metrics in coupling category is 14.2. The results illustrate that Dep_Out and Dep_In score 17 and 16 points, respectively. In Chapter 5, most of the respondents find the class relationship is an important criteria of classes that should be included in class diagrams. Therefore, if a class contains many dependencies, whether they are outgoing or incoming, this class is important. This could be the reason that many respondents stated that such a class should be included. EC_Attr has 15 points while IC_Attr has 17 points. If we compare the points between these two metrics and EC_Par (11 points)

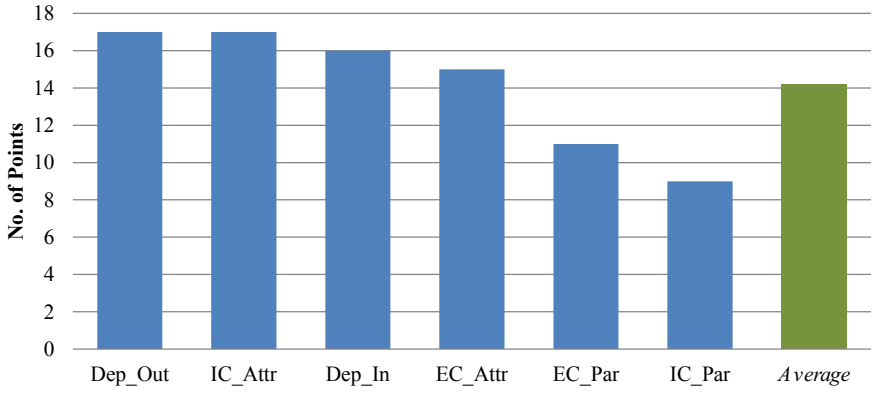


Figure 6.5: *Score Coupling Category (Question B5-B10)*

and IC_Par (9 points), there is a huge difference. This indicates that the classes that are declared and are used as an attribute are more important than the classes that are declared and are used as a parameter in class operations.

The Inheritance category consists of three metrics: NOC, DIT, and CLD. The average score for the metrics of this category is 10.7. From the results (Figure 6.6), NOC has the highest score in this category (20 points). DIT and CLD have 7 points and 5 points, respectively. These results suggest that the respondents may only be interested in a part of an inheritance hierarchy. If a class has a high NOC, the class is important since it has many immediate children and is also higher in the inheritance hierarchy. However, if a class has a high DIT, then this class is somewhere at the bottom of this hierarchy which means that there is a possibility that this class is not important. It is not a surprise that CLD has a low score because normally if a class has a high number of CLD then the class presents a very high-level of abstraction that is typically used to group the classes under this class.

Question B14 asked the reasons for including or excluding a class from the class diagram. The responses to open-ended question B14 were analyzed by grouping the answers into categories. An answer to this open-ended question could contain multiple keywords. The respondents stated that they wanted to include a class "when it is important" but they did not say when a class is important. It is a weakness of the survey-method that we could not ask for further questioning into more explicit factors when this answer was given.

Figure 6.7 shows the results of the question based on the keywords. It shows that there are three keywords that are related to the answers the most. These are Important/Relevant Class (29.6%), Domain Related (25.9%), and Coupling (18.5%). The keyword "Important/Relevant" is a broad term, but that is what the respondent answered. Hence, this answer is really obvious, but we cannot use it as a recipe to

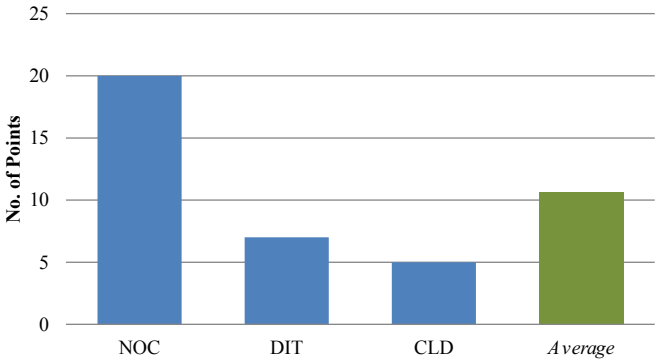


Figure 6.6: *Score Inheritance Category (Question B11-B13)*

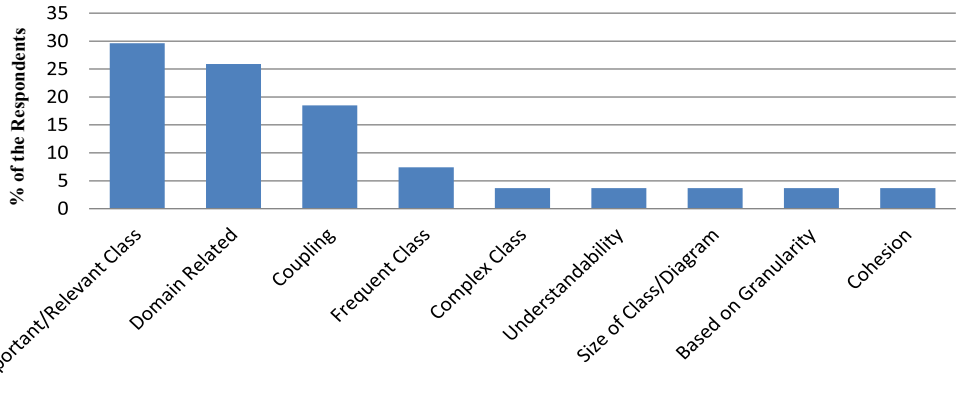


Figure 6.7: *Keywords to Include a Class in a Class Diagram*

decide which class is important for them. Coupling on the other hand is a factor that was expected. We stated in the survey in Chapter 5 that relationships are very important to understand a class diagram. Here, we found the same result. 18.5% of the respondents said that if a class has many relations, then that class should be included.

"Domain Related" are classes related to the concept or domain. Without these classes, it is hard for a software maintainer to understand a system. Five respondents mentioned the reasons for exclusion. One of these reasons is when a class is too small or that it can be combined with another class. Another respondent stated that he/she excludes a class if this class does not contain any important attributes or operations. Once again, the respondent did not state when an attribute or an operation is important. One respondent stated that he did not need any children classes. In other words, he only needs the parent classes. Another respondent mentioned that he would keep

the classes, but would exclude the attributes and operations from these classes to get a high-level abstraction. This answer is not really relevant to what we asked, but it is interesting to show the needs of the class names and class relationships to understand a system. The last respondent stated that he/she excludes helper classes or technical-specific classes since they are not needed to understand a system.

6.5.3 Practical Simplification Problems (Part C)

In this part, we tried to elicit characteristics about classes that should not be included in a class diagram. This information is gathered by asking the respondents to select classes that should be excluded from the class diagram of three actual system designs.

Coupling

In question C1, a class diagram of an ATM System (as shown in Figure 6.8) was presented without attributes and operations. Through this question, we aimed to elicit information about the influence of coupling category and class names. The overall results of this question are shown in Figure 6.9. The results show that 48% of the respondents chose to exclude the class Money and 36% of the respondents chose to exclude the OperatorPanel and Status class from the class diagram. We observed that these 3 classes have a relatively low coupling (≤ 2). Next, 32% of the respondents excluded the classes Deposit, EnvelopeAcceptor, ReceiptPrinter, Transfer and Withdrawal. The coupling of those classes is equal to 2. This means that the exclusion of 8 out of 24 classes of this class diagram could be explained based on their coupling. The classes that were important in this class diagram are Transaction and ATM. Only 4% of the respondents chose these classes as should not be included. Both classes have a high amount of coupling. This indicates that the amount of coupling plays a major role in selecting the classes that should or should not be included in a class diagram.

Meaningful Class Names

A RE-CD from a Library System (as shown in Figure 6.10) was used for question C2 ("*Figure 10*" in this questionnaire). All elements in a class diagram were presented (in HLoD) and we expected to discover the factors that are influential in selecting the classes that could be excluded. The results of the survey are shown in Figure 6.11.

From question C2, we found that class names also play a major role in determining whether a class should be included or excluded. The top three classes that were chosen to be excluded are AboutDialog, MessageBox and QuitDialog. From the class names, the respondents were able to predict what the functionality of the class is. AboutDialog, MessageBox and QuitDialog clearly referred to functionality that is used to display information. Thus, these classes are not considered important because they are only

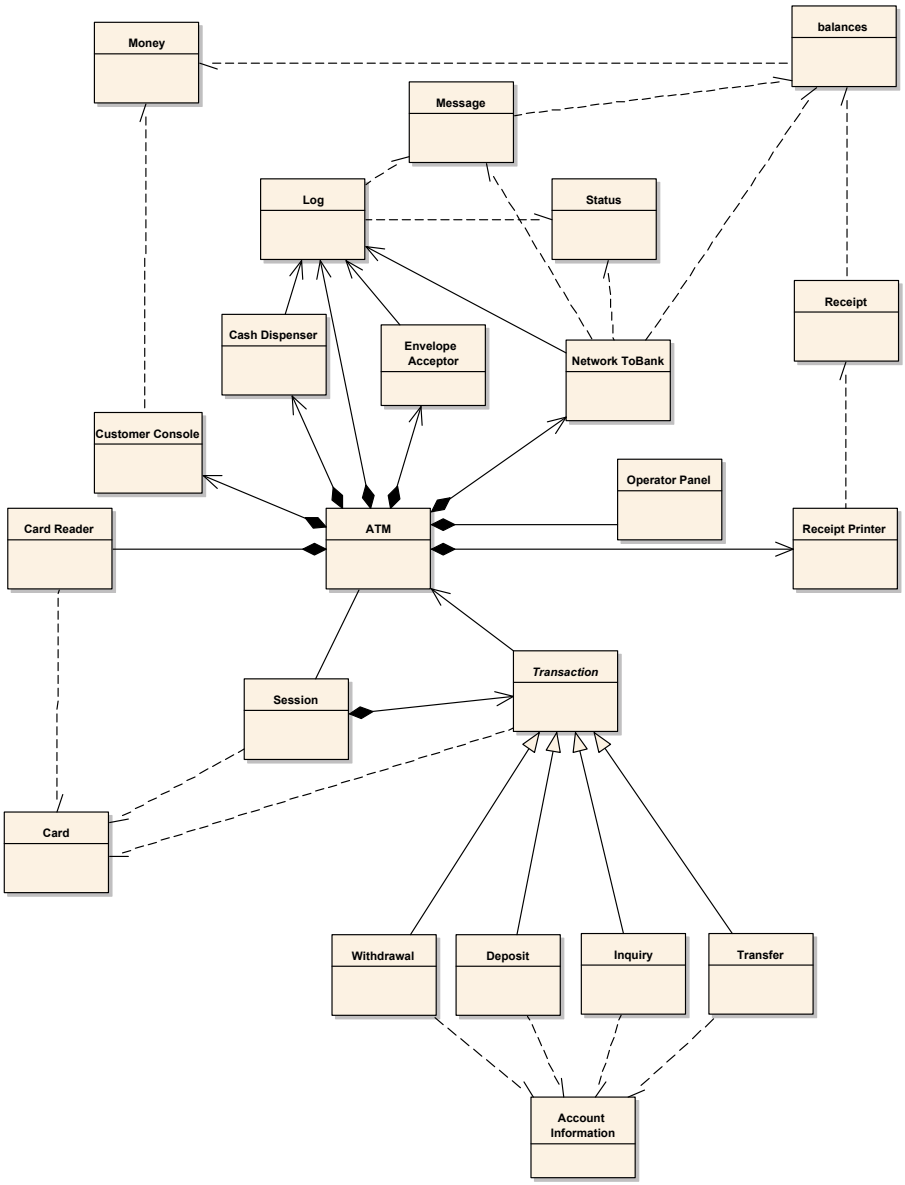


Figure 6.8: *Class Diagram A (ATM System)*

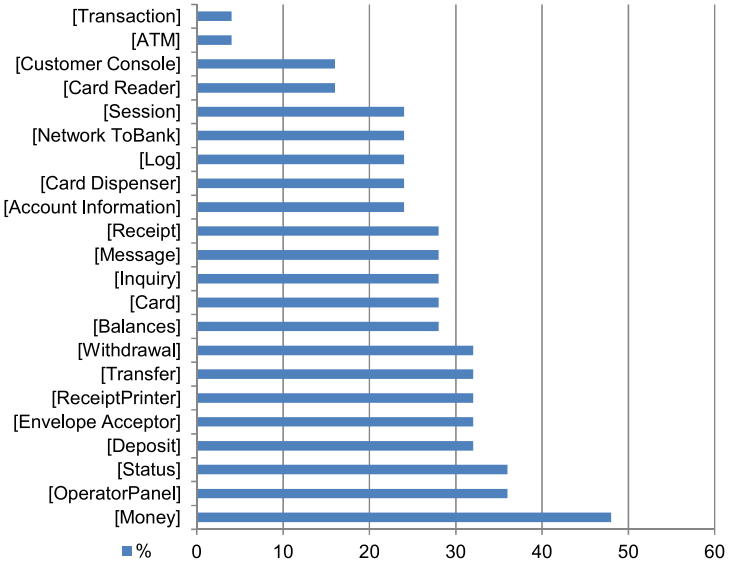


Figure 6.9: *Respondents Selection of Classes that Should not be Included in an ATM System*

used to display messages - which is not core functionality of this application. On the other hand, the 5 classes that not many respondents chose to exclude from the class diagram are classes that are related to the domain and have coupling more than 2. Borrower, Reservation, Loan, Item and Title are classes that have a meaningful name that might indicate the functionality of the classes and also are core concepts of the domain i.e. Library System.

Enumeration and Interface Classes

In question C3, the respondents were asked to select the classes that could be left out of a forward designed Pacman Game class diagram. Most of the classes in this diagram have relationships and meaningful class names. The complete result of this question is presented in Figure 6.12. The results indicate that 64% of the respondents chose to exclude class Direction from the class diagram. This class is an Enumeration class with coupling equals to 0 which might be the reason why this class should not be included in a class diagram. 52% of the respondents selected to exclude the Iterator class while 40% of the respondents chose to exclude the Iterable class. Both classes are interface classes that might indicate that those classes are not important or at least not related to the domain of the application. These results illustrate that the enumeration and interface types of classes are candidates for suppression in a simplifying this class diagram.

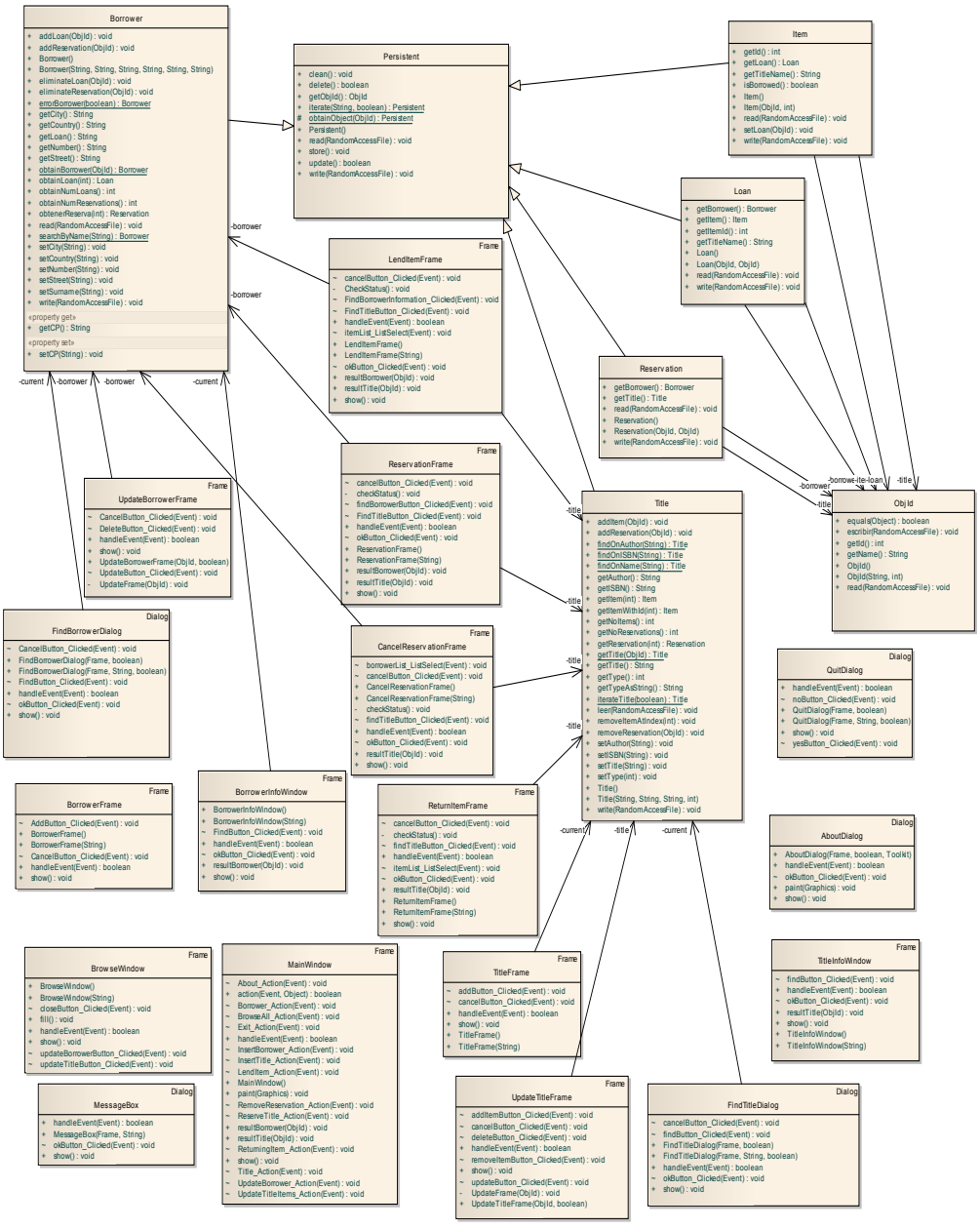


Figure 6.10: *Class Diagram B (Library System)*

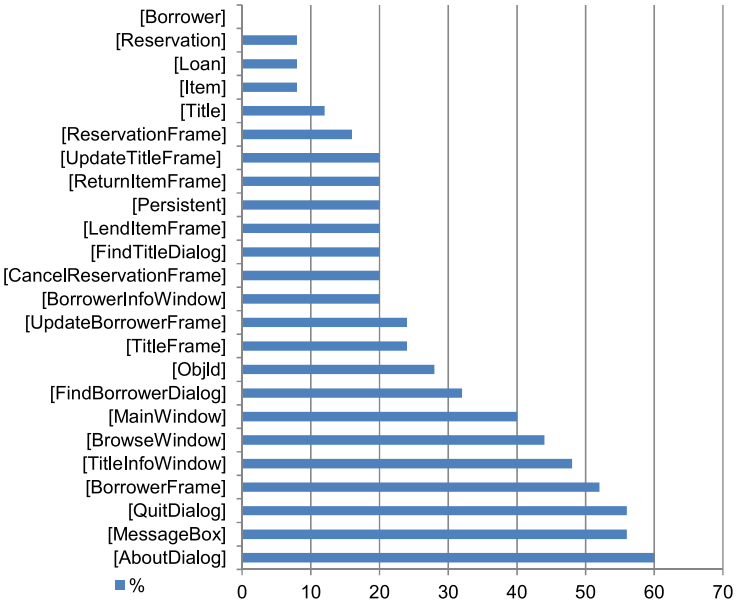


Figure 6.11: *Respondents Selection of Classes that should not be Included in a Library System*

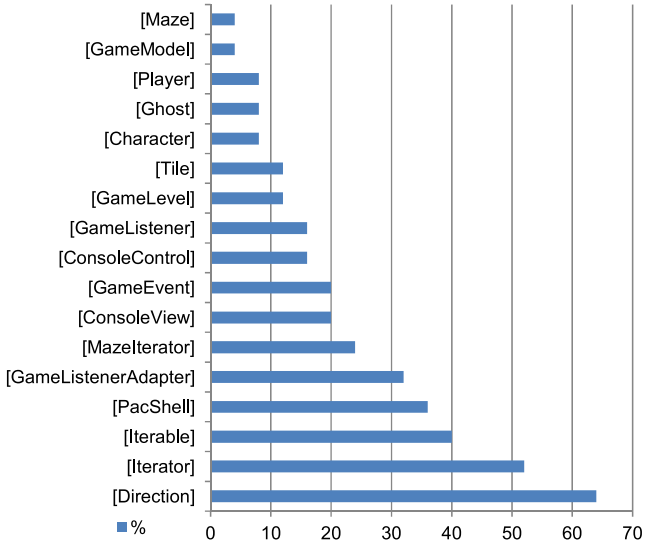


Figure 6.12: *Respondents Selection of Classes that should not be Included in a Pacman Game (Forward Design)*

Table 6.8: *The Preferences between Class Diagram A (C1), B (C2) and C (C3)*

Answers	Number of Respondents	in %	Respondent's Role			Respondent's Skill				
			Student	Researcher/ Academic	IT Pro.	Poor	Low	Avg	Good	Excellent
I prefer class diagram A (Figure 6.8)	5	20	0	2	3	0	0	2	1	2
I prefer class diagram B (Figure 6.10)	2	8	0	1	1	0	0	2	0	0
I prefer class diagram C (Figure 6.13)	12	48	6	5	1	1	3	6	1	1
I prefer them all	1	4	1	2	0	0	0	0	1	0
I do not prefer them	2	8	0	0	2	0	0	0	1	1
It does not matter which one	3	12	0	0	1	1	1	0	1	0
Total	**25**	**100**	**7**	**10**	**8**	**2**	**4**	**10**	**5**	**4**

Level of Detail

Referring to the class diagram A (question C1), class diagram B (question C2), and class diagram C (question C3), the respondents have been asked which flavour of the class diagram is preferred to be used. The detail information of LoD for the class diagrams is explained in Table 6.3.

The results in Table 6.8 show that almost half of the respondents (48%) preferred working with class diagram C (see Figure 6.13). This diagram is a HLoD forward design class diagram. They mentioned that the class diagram is clear, the necessary information is provided (e.g. attributes and operations) and most of the presented classes are important. This diagram was preferred by students and researchers and one IT Professional. 20% of the respondents preferred to use class diagram A. Most of the respondents that chose this diagram were Researchers/Academic and IT Professionals with the skill in class diagrams ranging from Average to Excellent. It seems that most of the respondents that have a good skill and experience in class diagrams prefer to use this diagram. The respondents mentioned that they preferred this diagram because it is simple, less technical, domain-oriented, systematic and has meaningful classes. 8% of the respondents preferred class diagram B. Another 8% did not prefer any of the

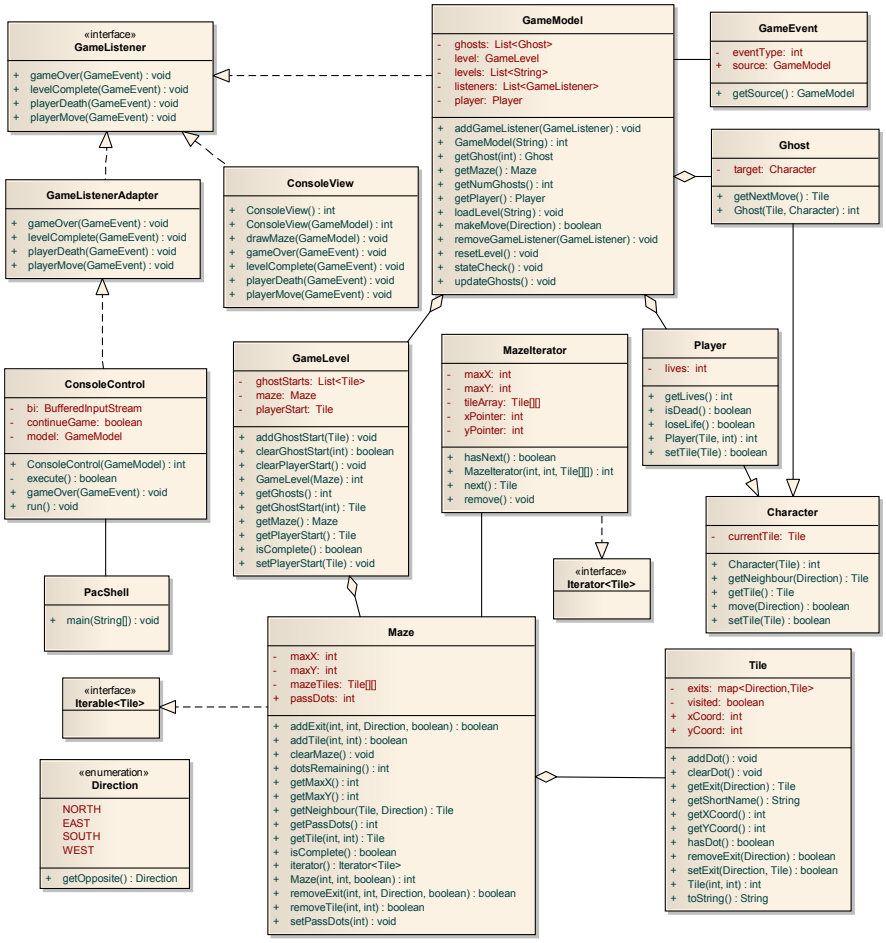


Figure 6.13: *Pacman Game Forward Design (Class Diagram C)*

presented class diagrams. As reason for this, they stated that there is "no story" in the class diagrams and that the class diagrams only show the solution, not the foundation of the domain.

Reverse Engineered Class Diagram which Conformed to Forward Design

In question C5, the class diagram used was slightly different from the class diagram presented in question C3 because this class diagram was constructed by using a reverse engineering technique (see Figure 6.14). In this question, we tried to discover if there was any difference in selecting the classes that should not be included in a class diagram in a Re-CD that is close or almost similar to the forward design class. The result shows that the class Direction and PacShell were selected by 72% of the respondents to be left

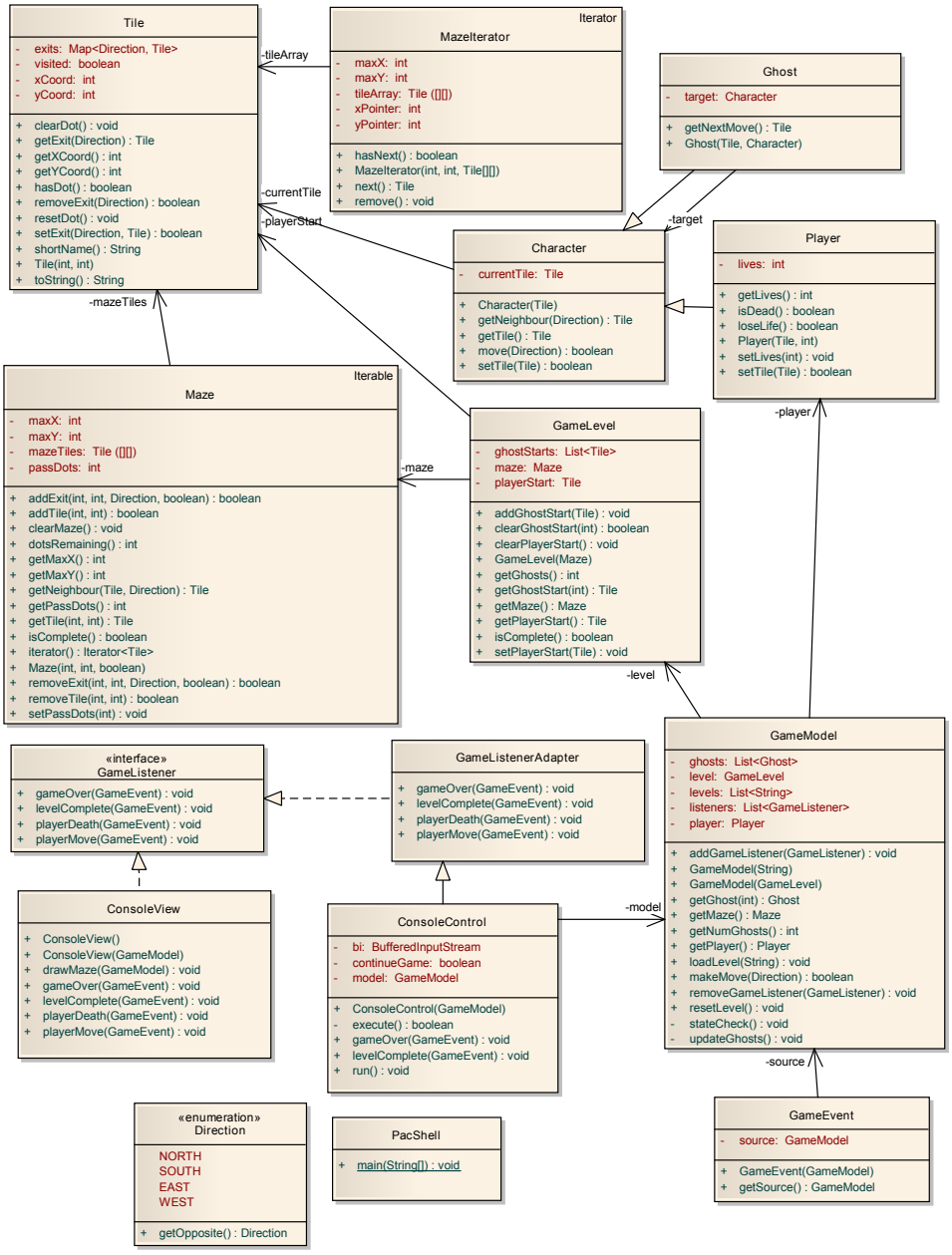


Figure 6.14: *Reverse Engineered Pacman Game (Class Diagram D)*

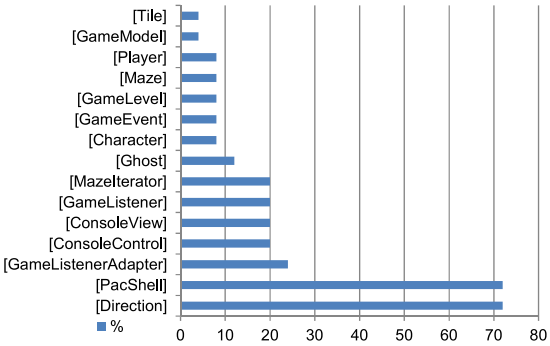


Figure 6.15: *Respondents Selection of Classes that should not be Included in a Pacman Game (Reverse Engineered Design)*

out from the class diagram. Compared to question C3, the Iterator and Iterable classes were differently presented in this reverse engineered diagram. The complete result of question C5 is shown in Figure 6.15.

Reverse Engineered vs. Forward Engineer Class Diagram

Question C6 aimed to discover which type of class diagrams was preferred by the respondents i.e. RE-CD or forward designed class diagrams. The RE-CD used in this question was different from the RE-CD in question C2 because this RE-CD was derived from a system that was implemented (or coded) closely with the forward design. The results in Table 6.9 show that most of the respondents (mainly researcher) preferred the class diagram D (the RE-CD from question C5 as illustrated in Figure 6.14). 40% of the respondents chose this diagram because it is more detailed, clear, interface classes and it is easier to understand. 20% of the respondents did not choose any of the two class diagrams because they did not have a preference. The reason mentioned by these respondents was that both class diagrams are equally good and similar. On the other hand, 16% of the respondents preferred the class diagram C (from question C3). There is no pattern of selection present in this result in terms of the respondents' role and skill.

If we compare the results of this question and the results of Section 6.5.3, we found that RE-CD is chosen if the source code was closely following the forward design, because then the difference between forward design and its RE-CD is small.

6.6 Discussion

In this section, we discuss the results and findings presented earlier in this chapter. This section is divided into 5 parts which are: Respondents' Background, Software

Table 6.9: *The Preference between Class Diagram C and D*

Answers	Number of Respondents	in %	Respondent's Role			Respondent's Skill				
			Student	Researcher/ Academic	IT Pro.	Poor	Low	Avg	Good	Excellent
I prefer class diagram D (Figure 6.13)	10	40	1	7	2	1	1	5	2	1
I prefer class diagram C (Figure 6.14)	4	16	1	0	3	0	1	1	1	1
I prefer them both	3	12	1	2	0	0	0	2	1	0
I don't prefer them	3	12	2	1	0	0	2	0	0	1
It doesn't matter which one	5	20	2	0	3	1	0	2	1	1
Total	**25**	**100**	**7**	**10**	**8**	**2**	**4**	**10**	**5**	**4**

Design Metrics, Class Names and Coupling, Class Diagrams Preferences, and Threats to Validity.

6.6.1 Respondents' Background

In Part A, the respondents' profiles to this questionnaire were quite evenly distributed. The location of the respondents showed that most of the respondents are from The Netherlands and Malaysia. This is due to the professional and personal network of the author.

In terms of the respondent's skill and experience with class diagrams, we found that 72% of the respondents have more than 1 year of experience with UML and that 76% of the respondents have rated themselves average or above if it comes to creating, modifying and understanding class diagrams. Even though 28% of the respondents said that they had less than one year of experience, we can still state that all the respondents have knowledge about class diagrams.

6.6.2 Software Design Metrics

We asked which metrics could be used as indicators for inclusion or exclusion of classes from class diagrams. We classified metrics in 3 categories: Size, Coupling and Inheritance. The overall ranking of the score is shown in Table 6.10.

Table 6.10: *Overall Score for Software Design Metrics*

No.	Design Metrics	Score
1	NumPubOps	25
2	NOC	20
3	NumOps	18
4	NumAttr	17
5	Dep_Out	17
6	IC_Attr	17
7	Dep_In	16
8	EC_Attr	15
9	Setters/Getters	13
10	EC_Par	11
11	IC_Par	9
12	DIT	7
13	CLD	5

In the Size category, we found that the higher number of public operations, is the more people prefer this class. There is a clear preference for public operations over operations in general because public operations are considered to more often stand for important functionality.

In the Coupling category, we found that the classes that have many incoming and outgoing dependencies are preferred for inclusion. We found that IC_Attr and EC_Attr have higher scores than EC_Par and IC_Par. The reason might be that a class that is declared as an attribute is more important because it can be used across many operations in the class.

In the Inheritance category, we discovered that for a class that has a high NOC, the class should be included in a class diagram. This parent class is helpful to show the abstraction of a group of classes. For DIT, a higher number of DIT does not indicate it is an important class because it basically means that this particular class is located very low in the inheritance hierarchy which means that this class is too detailed and most of the times not needed. For CLD, if a class has a high value for this metric this that this class is very abstract, meaning that this class alone will not be enough to understand the whole hierarchy.

As for the complete results, we found that NumPubOps has the highest points across all metrics. Also, all the metrics received a positive score, even though some only slightly, while a negative score is also possible. Hence, each of these can be considered in our subsequent studies on how to best use these metrics in selecting classes for in/exclusion.

6.6.3 Class Names and Coupling

In Part C, we showed that coupling is a highly influential factor when we are trying to exclude classes from a class diagram. Another influencing factor is the class name. From our observation, the class names are an influencing factor since it may indicate the class role and responsibility. Through class names, the respondents make assumptions of the class functionality (role, responsibility, service) and as well as the flow of the system. Many respondents excluded Graphical User Interface (GUI) related classes in the Library system because of the class role and responsibility (class name based), and coupling.

Aside from these two big influencing factors, many respondents excluded types of classes like enumeration and interface. Either of these classes did not contain any information in it or the coupling was very low. Another reason of why the interface classes are excluded could be that these classes are generic and not key for the domain of an application. Overall, we suggest that the role and responsibility of the classes (based on class names) could be an influencing factor in deciding class inclusion/exclusion.

6.6.4 Class Diagram Preferences

In question C4, we found that most of the respondents preferred to use HLoD of the forward design. The reasons the respondents gave was that this class diagram is clear and the necessary information is provided in this class diagram. Meanwhile, in question C6, most of the respondents had chosen the RE-CD (HLoD). The reason might be that the RE-CD that was provided has few differences from the forward design. The respondent stated that they preferred this diagram because they find it more detailed and it is easier to understand. Some of the respondents also mentioned that the interface classes are removed, which make it a better class diagram.

From our observation, the RE-CD of the Library system was not preferred because the structures of the classes were not well-presented. This might be because the implementation was not conforming to the design or there was no design in the system before implementation.

6.6.5 Threats to Validity

Although the respondents of this survey were quite well distributed between the status roles (Student, Researcher/Academic and IT Professional), we consider that the amount of full responses was still a small number. The locations of the respondents were biased to The Netherlands and Malaysia. Most of the questions in this study require the respondent to choose the best answers. We have made several assumptions about why the respondents chose these answers and these assumptions may not be accurate.

6.7 Conclusion

In this survey, we revealed the metrics that indicate the classes that could be left out. We also found the flavour of class diagrams that developers prefer to work with. From the results, we found that the most important software design metric is the Number of Public Operations. This means that if a class has a high number of public operations then this indicates that this class is important and should be included in a class diagram. In this survey, we also found that the class names and coupling are influencing factors when selecting a class to be excluded from a class diagram.

With these results, we can highlight which classes should be included or excluded in RE-CDs. This is based on our results and analysis by looking at the metrics and behaviour the respondents had in Part C. Although the number of responses on this questionnaire is not that high, we managed to find some influencing factors for deciding on class-inclusion or exclusion from a class diagram.

6.8 Future Work

This chapter reports an early study on how to simplify class diagrams and we see a number of ways to extend this work. We propose to validate the results by using an industrial case study and discover the suitability of the simplified class diagram for the practical usage. It would also be interesting to include other metrics that we have not chosen and check whether they are important or not and ask why the respondent chose the answer to get the reason.

From the results, we found that class role and responsibility are important indicators in a class diagram. We would like to suggest a study on the names (class, operation and attribute) that the software developers find important or meaningful in order to understand a system. We discovered some weakness in the questionnaire and our suggestion is to improve this questionnaire by increasing the amount of responses. It would be interesting to see what the results are with a larger group of respondents.

Chapter 7

Condensing Reverse Engineered Class Diagram using Object-Oriented Design Metrics

There is a range of techniques available to reverse engineer software designs from source code. However, these approaches generate highly detailed representations. The condensing of reverse engineered representations into more high-level design information would enhance the understandability of reverse engineered diagrams. This chapter describes an automated approach for condensing reverse engineered diagrams into diagrams that look as if they are constructed as forward designed UML models. To this end, we propose a machine learning approach. The training set of this approach consists of a set of forward designed UML class diagrams and reverse engineered class diagrams (RE-CD for the same system). Based on this training set, the method learns to select the key classes for inclusion in the high-level class diagrams. In this chapter, we studied a set of nine classification algorithms from the machine learning community and evaluated which algorithms perform best for predicting the key classes in an application.

7.1 Introduction

Up-to-date design documentation is important, not just for the initial design, but also in later stages of development and in the maintenance phase. UML models created

This chapter is adapted from a publication entitled **"An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams"**, In Proceedings of the International Conference on Software Maintenance (ICSM 2013)

during the design phase of a software project are often poorly kept up-to-date during development and maintenance. As the implementation evolves, correspondence between design and implementation degrades [118]. For legacy software, faithful designs are often no longer available, while these are considered valuable for maintaining such systems.

A popular method to recover an up-to-date design of a system is reverse engineering. Reverse engineering is the process of analyzing the source code of a system to identify the system's components and their relationships and create design representations of the system at a higher level of abstraction [41]. Reverse engineering also refers to methods aimed at recovering knowledge about a software system in support of execution some of software engineering tasks [170]. Tool support during maintenance, re-engineering or re-architecting activities has become important to decrease the time that software personnel spends on manual source code analysis and helps to focus attention on important program understanding issues [22]. However, current reverse engineering techniques do not yet solve this problem adequately. In particular, RE-CDs are typically a detailed representation of the underlying source code. This makes it difficult for software engineers understand what the key elements in the software structure are [125].

This study is partially motivated by a scenario when new programmers want to join the development team. They need a starting point in order to understand the whole system before they are able to modify it. Provided with the software design, the programmer will normally browse the class design and try to synchronize the design with the source code. There is a need for programmers recognize which classes in the system play important roles or can be considered as key classes in the system.

Fernández-Sáez et al. [59] found that developers experience more difficulties in finding information in reverse engineered diagrams than in forward designed diagrams and also find the level of detail in forward designed diagrams more appropriate than in reverse engineered diagrams. In order to achieve better reverse engineered representations, we need to learn which information from the implemented system to include and which information to leave out. A method to assist software engineers to focus on the key classes and aspects of the design is needed. The identification of key classes can also be used to simplify complex class diagrams or help to predict the severity of a defect in a software system.

This study specifically aims at providing suitable classification algorithms to decide which classes should be included in a high-level class diagram. We seek an automated approach to classifying the key classes in an application. We require algorithms that are able to produce a score, not just a classification, so that a user potentially has the option to choose a particular level of abstraction for representing a reverse engineered design (in particular RE-CD).

In this chapter, we focus on the use of design metrics as predictors (input variables used by the classification algorithm). The advantage of using design metrics is that these can be obtained very efficiently with little effort. This fits our objective of creating

a method of practical use to software developers. Also, we analyse the predictive power of the predictors to know how influential each of these predictors are, with respect to the performance of the classifier.

We explore several classification algorithms for predicting key classes that should be included in a class diagram. As the input for this study, we use sets of source codes from the open source projects with corresponding UML models that contain forward designed class diagrams. We use these class diagrams as 'ground truth' to validate the quality of the prediction algorithms. The methods we study will 'learn' from the forward designed class diagrams which classes should be selected from the RE-CDs. In total, nine algorithms were selected for this comparison study. These algorithms will be evaluated in terms of accuracy and robustness with respect to the information that they recommend to keep in and leave out of the class diagram. The candidate set of algorithms includes: J48 Decision Tree, k-Nearest Neighbor (k-NN), Logistic Regression, Naive Bayes, Decision Tables, Decision Stumps, Radial Basis Function Networks, Random Forests and Random Trees.

We have collected a diverse collection of data sets consisting of nine pairs of UML design class diagrams and associated Java source code derived from open source software projects. The number of classes in the source code of these projects ranges from 59 to 903. Out of these classes, 3% to 47% were found to be included in the forward UML class diagram. The open source projects were chosen for a number of reasons. We wanted the data to be representative for the diversity and complexity of real world projects. The quality of documentation for open source projects varies widely, and there is also a substantial variation in the ratio of classes in the forward design versus classes in the source code. In open source projects, software design is not a mandatory requirement, and these projects rely on volunteers working together in a distributed fashion. Also, by using open source projects we make it easier for other researchers to reproduce or compare against our results and develop new methods on the same data.

The chapter is structured as follows: Section 7.2 discusses related research and Section 7.3 describes the research questions. Section 7.4 explains the approach on how we conducted the evaluation. Section 7.5 presents the results and Section 7.6 discusses our findings. This is followed by conclusions and future work in Section 7.7.

7.2 Related Work

The following studies are related to our research from the perspective of identifying key classes from software artifacts.

Zaidman and Demeyer [184] proposed a method for identifying key classes by using Hyperlink-Induced Topic Search (HITS) web mining technique. They used dynamic (runtime) analysis of source code as the input for the identification of key classes. For validating their method, they manually identified key classes from the

software documentation. Recall and precision were used to evaluate the approach and they found that their approach was able to recall 90% of the key classes and the precision was slightly under 50%. However, dynamic analysis approaches need significant effort for collecting run-time traces.

Perin et al. [136] proposed ranking software artifacts using the PageRank algorithm. They used the Pharo Smalltalk system and Moose re-engineering environment as case studies. For the Pharo Smalltalk system, they reported that their approach was able to detect 42% of the important classes (prediction based on classes) and to detect 52% of the important classes when prediction was based on methods. However, no precision result was presented for the Moose system.

Hammad et al. [77] proposed an approach to identify the critical software classes in the context of design evolution. Version (commit) history and source code were used as the input for this study. They assumed that the classes that were frequently changed in the software evolution are the classes that are critical to the system. They found that 15% of the classes in the case studies were changed from six design changes.

Steidl et al. [154] presented an approach to retrieve important classes of a system by using network analysis on the dependency graphs. They performed an empirical study to find the best combination of centrality measurement and dependency graphs. Classes recommended by their test project developers were used as the baselines. They found that the centrality indices perform best when using the undirected dependency graph that include information about inheritance, parameter and return dependency.

Our work also aims at identifying key classes, but we explore diverse classification algorithms based on supervised machine learning. In contrast, static analysis is used for our data collection as it is easy to obtain from open source projects. The aforementioned works validate their approach to identify the key classes in class diagrams by comparing the prediction result with the information derived from software documentation, repository and developer(s) recommendation. In this study, we validate our approach by comparing selected classes against those actually found in the forward design.

7.3 Research Questions

This section describes the research questions of this study that will be answered in Section 7.5.

RQ1: Which design metrics are influential predictors in classifying key classes?
For each case study, we explore the predictive power of individual predictors.

RQ2: How robust is the classification to the inclusion of categories of predictors?
We explore how the performance of the classification algorithms is influenced by partitioning the predictor-variables in different groups with different characteristics.

RQ3: What are suitable classification algorithms in classifying key classes?
The candidate classification algorithms are evaluated to find out which algorithm(s) are suitable for classifying the key classes in a class diagram.

7.4 Approach

This section describes the Examined Predictors and Tools, Case Studies and Process.

7.4.1 Examined Predictors and Tools

In this section, we describe i) the metrics that we used as inputs to the prediction algorithms, and ii) the tools used for this research.

Examined Predictors

We used a set of software metrics that are typically used to characterize design characteristic of classes in class diagrams as input to our classification algorithms. The SDMetrics [180] tool is capable of computing 32 types of class diagram metrics. These metrics are divided into five categories, namely Size, Coupling, Inheritance, Complexity and Diagram. We select 11 class diagram metrics from the Size and Coupling category. These categories of metrics were selected for the following reasons: i) our survey in Chapter 5 and 6 demonstrated that developers use Size and Coupling as predictors of key classes, ii) experts in the area of software metrics (Briand et al. [36] and Genero et al. [29]) stated that Coupling is an important structural dimension in object-oriented design, iii) the work in [188] and [73] showed that WMC (a metric in the Size category) and CBO (a metric in the Coupling category) are influential for defect prediction. The specific set of 11 metrics used is shown in Table 7.1.

Tools

The tools used in this study are the following:

- *Reverse Engineering Tool*: MagicDraw[9] is a system modeling tool that provides reverse engineering facilities. MagicDraw version 17.0 (academic evaluation license) was used for this study.
- *Software Metrics Tool*: SDMetrics is a tool that computes a large set of metrics for UML models. SDMetrics version 2.2 (academic license) was used for this study.
- *Data Mining Tool*: Waikato Environment for Knowledge Analysis (WEKA) is a collection of machine learning algorithms for data mining tasks [178]. It contains tools for data pre-processing, classification, clustering, and visualization. WEKA version 3.6.6 was used for this study.

Table 7.1: *List of Class Diagram Metrics*

Metrics	Category	Description
NumAttr	Size	The number of attributes in the class.
NumOps	Size	The number of operations in the class. Also known as WMC in [40] and Number of Methods (NM) in [99].
NumPubOps	Size	The number of public operations in a class. Also known as Number of Public Methods (NPM) in [99].
Setters	Size	The number of operations with a name starting with 'set'.
Getters	Size	The number of operations with a name starting with 'get', 'is', or 'has'.
Dep_Out	Coupling (import)	The number of dependencies where the class is the client.
Dep_In	Coupling (export)	The number of dependencies where the class is the supplier.
EC_Attr	Coupling (export)	The number of times the class is externally used as attributes type. This is a version of OAEC +AAEC in [34].
IC_Attr	Coupling (import)	The number of attributes in the class have another class or interface as their type. This is a version of OAIC+AAIC in [34].
EC_Par	Coupling (export)	The number of times the class is externally used as a parameter type. This is a version of OMEC+AMEC in [34].
IC_Par	Coupling (import)	The number of parameters in the class have another class or interface as their type. This is a version of OMIC+AMIC in [34].

7.4.2 Case Studies

We used the following criteria for selecting case studies:

- The software should be an open source software project that provides both the implementation source code and forward design class diagram.
- The number of classes in the implementation (source code) ≥ 50 classes.

Based on these criteria, nine open source software/systems were selected. In these projects, we selected a forward UML class diagram from the documentation and then selected a matching version of the source code. The number of classes in these case studies ranges from 59 to 903 (see Table 7.2). The ratio between the number of classes included in the UML class diagram and the number of classes in the implementation

Table 7.2: *List of Case Study*

No.	Project	Total Classes in Source Code (S)	Total Classes in Design (D)	D:S ratio as %
1	ArgoUML	903	44	4.9
2	Mars	840	29	3.5
3	JavaClient	214	57	26.6
4	JGAP	171	18	10.5
5	Neuroph 2.3	161	24	14.9
6	JPMC	121	24	19.8
7	Wro4J	87	11	12.6
8	xUML	84	37	44.1
9	Maze	59	28	47.5

(source code) spreads across a wide range: from 3 to 47%. This large range in characteristics of the input may be a difficulty for building a reliable classifier for our domain. For this reason, we focus on algorithms that will produce a score for a class concordant with the likelihood that it would be included in the UML diagram. This will allow a developer to vary the amount of classes included, i.e. the level of abstraction, by changing the threshold on the score. We make the case studies used for this research available at [6] for future research and for validation of this study. Detailed information about these case studies can be found in Section 3.3.

7.4.3 Process

This subsection describes the steps performed for this study. The inputs for this process are forward designs and RE-CDs (constructed from the source code of the case studies). The output of the machine learning phase is the list of key classes that can be used to condense a class diagram. The approach is illustrated in Figure 7.1.

Data Preparation

For data preparation, class diagram metrics were extracted from RE-CD (obtained through reverse engineering process). Then, the information about the presence of a class in the forward design was entered in a table.

The data preparation steps are described in Table 7.3. In this table, Step 1 to 3, performed the extraction of all required data. Then, the data is cleaned up by removing the external library and runtime classes since we only focused on the application and domain related classes (as suggested in Chapter 5). Step 5 and 6 perform the merging of data and Step 7 select the software metrics that are useful for prediction.

We expect that there is some noise in the predictors. For instance, the getters and

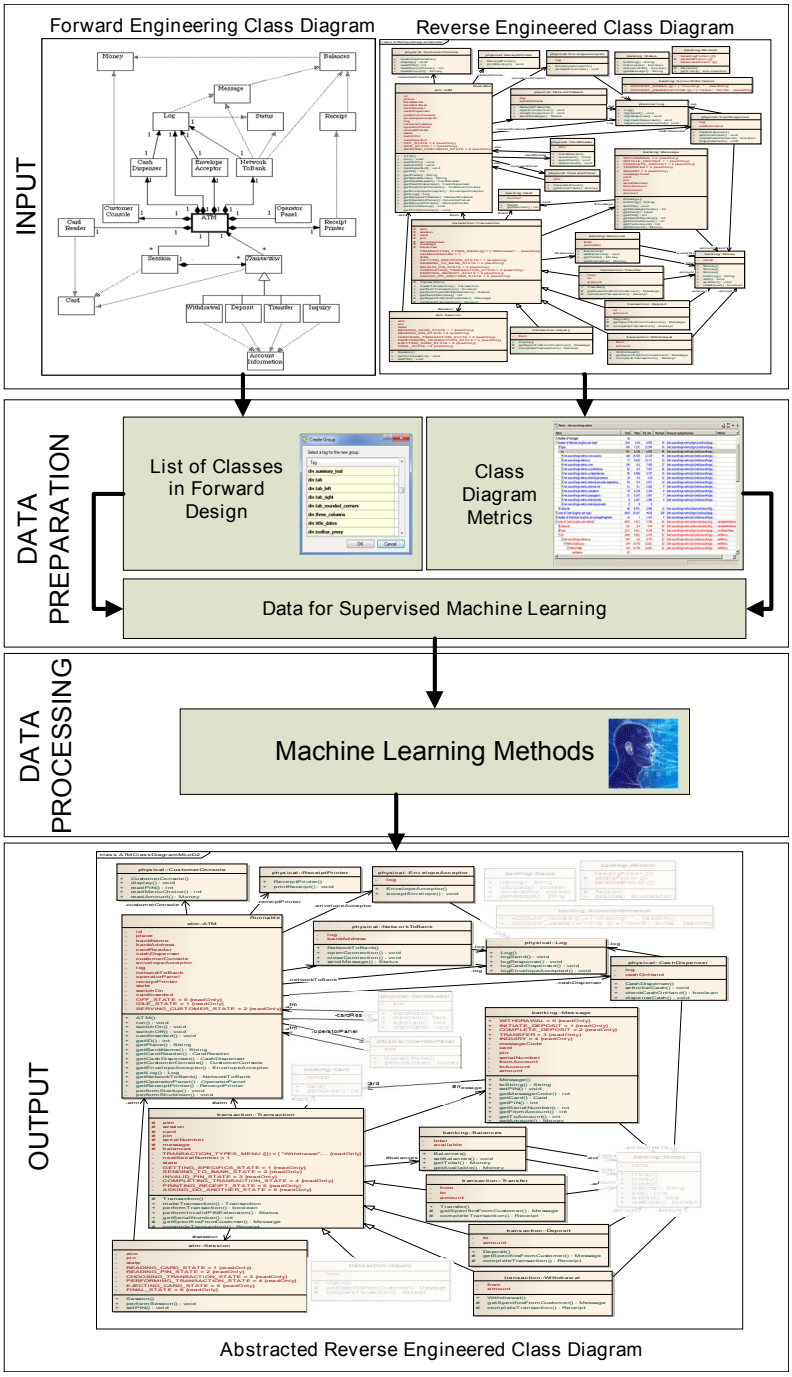


Figure 7.1: *Design Abstraction Process*

Table 7.3: *Data Preparation Steps*

No.	Preparation Step	Description
1.	List all the classes that appear in the UML design-class diagram	To get the class in design vs. implementation ratio
2.	Reverse engineer the source code into a class diagram using MagicDraw. Save the class diagram in XML Metadata Interchange (XMI) file format	To get the design from the source code prepared for the metrics tool input
3.	Calculate the software metrics of the RE-CD using SDMetrics and save in CSV format	Class diagram metrics calculation and data mining input preparation
4.	Manually remove external library classes and runtime classes from the list	To extract only developed classes in the source code
5.	Merge the software metrics information from the source code and the classes in the forward design	To map between classes in design and classes from software metrics obtained from the source code
6.	Amend the CSV file by adding the information of "In Design" properties (N for not presented in Design Document, Y for presented in Design Document)	Add the information about the class in design in the software metrics information
7.	Remove software metrics properties that show no significant information (data cleanup) present an overall data summary in plot graph	To extract only the selected independent variable (class diagram metrics) and present the summary of data

setters predictor completely relies on the conformance of source code to naming conventions (e.g. 'get','has'). Not all case studies have this kind of naming convention. In RQ2, we desired to explore the performance of the classification algorithms across different group of predictors (based on predictors' characteristics). Therefore, we experimented with different groups of predictors: Experiment A: the full set of predictors (predictor set A), Experiment B: all metrics, but excluding metrics related to getters, setters and Number of Public Operation (predictor set B), and Experiment C: a set of predictors that only uses Coupling metrics (predictor set C). The details of all predictor sets are shown in Table 7.4.

Data Processing

For every run of the classification algorithm, we randomly split the dataset (for every case study) into 50% for the training set and the other 50% for the test set. To further

Table 7.4: *Predictor Sets*

No.	Predictor	Predictor set A	Predictor set B	Predictor set C
1	NumAttr	Yes	Yes	No
2	NumOps	Yes	Yes	No
3	NumPubOps	Yes	No	No
4	Setters	Yes	No	No
5	Getters	Yes	No	No
6	Dep_out	Yes	Yes	Yes
7	Dep_In	Yes	Yes	Yes
8	EC_Attr	Yes	Yes	Yes
9	IC_Attr	Yes	Yes	Yes
10	EC_Par	Yes	Yes	Yes
11	IC_Par	Yes	Yes	Yes

improve reliability, we ran each experiment 10 times using different randomization. The main reason for doing this is that the data are typically imbalanced where the number of classes in design (the 'positives') is very low compared to the number of classes in the source code. If we would have used 10-fold cross validation, it means that we use only 10% of the data for testing and 90% for training. Thus, the possibility of any positives to be included in the test data was very low, and test set error measurements would not have been reliable (refer [62] for more detail discussion). For example, let say we have an imbalanced dataset with 900 examples with only 1% (9) of the examples are positive. If we used 10-fold cross validation, there is a possibility of the positive example is not included in the test set. Thus, the True Positive Rate (TPR) calculation is not reliable in this situation. We avoided detailed fine-tuning because we assumed our end users have no knowledge of data mining. Algorithms ran with default WEKA configuration. We used WEKA as the tool and algorithms ran with WEKA default parameter setting; Except for k-Nearest Neighbor, for which we used two different neighborhood size settings (1 and 5 neighbors). A different number of k in k-NN may present a substantial difference in classification performance. Therefore, this experiment investigates two sets of k-NN: (a) k=1 (extreme lowest value of k, or plain nearest neighbor); and (b) k=5 (which we believe it represents a more average k value for the dataset).

Evaluation

In this study, the analyses are conducted using two evaluation measures: i) the univariate analysis, and ii) the analysis of classification performance. These measures are explained as follows:

Univariate Analysis: To measure the predictive power of the predictors, we use the information gain with respect to the class [76]. Univariate predictive power means measuring how influential a single predictor is in predicting performance. The results of this algorithm are normally used to select the most suitable predictor. Nevertheless, in our study, we did not use it for predictor selection, but for an exploratory analysis of the usefulness of various predictors (in this case: class diagram metrics).

For Univariate analysis, the predictors were evaluated by using the Information Gain (InfoGain) Attribute Evaluator in WEKA. This method produces a value which indicates the influence of a predictor in prediction performance based on the case studies. A higher value of InfoGain denotes a stronger influence of the predictor (i.e. closer to 1 is better).

Analysis of classification performance: As discussed in Chapter 2, classification performance is analyzed by using the Area Under ROC Curves (AUC). The evaluation of machine learning classification algorithms started with generating a confusion matrix (as shown in Table 2.3), based on applying a classification algorithm using WEKA.

WEKA provides AUC calculations as a number between 0 and 1. A value closer to 1 means a better classification result, while a value close to 0.50 means the classification performs almost randomly.

Based on an early observation on our case studies, we decided the threshold for the AUC value = 0.60. This means, if the AUC value $\geq$ 0.60, the classification algorithm is considered to be usable for prediction for our specific problem.

7.5 Evaluation of Results

This section presents our evaluation on i) predictive power of predictors and ii) overview of benchmark AUC results.

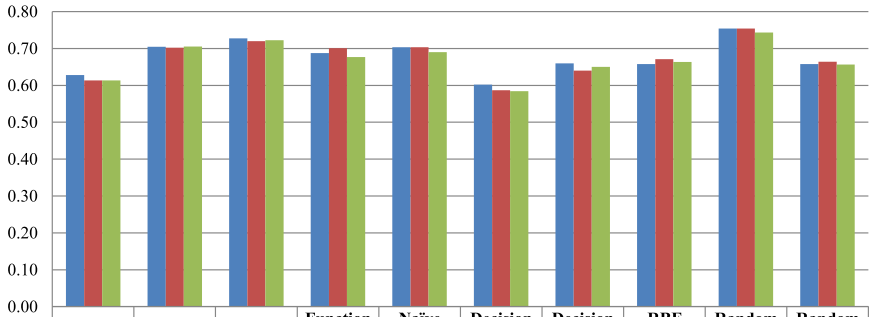
7.5.1 Predictor Evaluation

This subsection presents our univariate analysis results that measure the predictive performance of each predictor using information gain.

RQ1. The results show the influence of a predictor for the classification algorithm. A class diagram metric is considered to be influential for prediction when the value of the InfoGain Attribute Evaluator is greater than 0. Table 7.5 shows that out of eleven class diagram metrics used in this study, nine of them influenced the prediction in the JavaClient project while xUML and Mars have eight and seven influential class diagram metrics. On the other hand, ArgoUML and JPMC have only one influential class diagram metric. Table 7.5 also shows that the Coupling category metrics are influential for every case study. In all cases, at least one of the Coupling metrics is

Table 7.5: *Univariate Predictor Performance (Information Gain)*

Project	NumAttr	NumOps	NumPubOps	Setters	Getters	Dep_out	Dep_In	EC_Attr	IC_Attr	EC_Par	IC_Par
ArgoUML	0.000	0.000	0.000	0.000	0.000	0.024	0.000	0.000	0.000	0.000	0.000
Mars	0.000	0.013	0.017	0.011	0.025	0.000	0.047	0.037	0.000	0.031	0.000
JavaClient	0.093	0.048	0.044	0.000	0.050	0.215	0.093	0.000	0.183	0.092	0.225
JGAP	0.073	0.056	0.000	0.078	0.000	0.047	0.000	0.000	0.000	0.058	0.000
Neuroph	0.000	0.054	0.062	0.000	0.000	0.000	0.084	0.000	0.000	0.106	0.000
JPMC	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.059	0.000	0.000
Wro4J	0.000	0.000	0.000	0.000	0.000	0.000	0.212	0.111	0.000	0.196	0.000
xUML	0.168	0.281	0.281	0.306	0.147	0.240	0.000	0.000	0.085	0.000	0.506
Maze	0.000	0.000	0.000	0.000	0.000	0.000	0.171	0.178	0.000	0.125	0.000
No. of InfoGain > 0	**3**	**5**	**4**	**3**	**3**	**4**	**5**	**3**	**3**	**6**	**2**
Average	**0.037**	**0.050**	**0.045**	**0.044**	**0.025**	**0.058**	**0.067**	**0.036**	**0.036**	**0.068**	**0.081**



	J48	k-NN(1)	k-NN (5)	Function Logistic	Naïve Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
Predictor Set A	**0.63**	**0.70**	**0.73**	**0.69**	**0.70**	**0.60**	**0.66**	**0.66**	**0.75**	**0.66**
Predictor Set B	**0.61**	**0.70**	**0.72**	**0.70**	**0.70**	**0.59**	**0.64**	**0.67**	**0.75**	**0.66**
Predictor Set C	**0.61**	**0.71**	**0.72**	**0.68**	**0.69**	**0.58**	**0.65**	**0.66**	**0.74**	**0.66**

Figure 7.2: *Average AUC Score for Every Dataset.*

listed as influential for prediction. This means that class diagram metrics categorized in Coupling (i.e. IC_Par, EC_Par, IC_Attr, EC_Attr, Dep_In and Dep_out) have a strong influence on prediction ability. If we compare Coupling metrics with Size metrics (i.e. NumAttr, NumOps, NumPubOps, Getters, Setters) we found that only five case studies listed at least one of the Size-metrics as influential predictor. EC_Par is the most influential class diagram metrics because it is listed as influential in prediction for six out of nine case studies.

RQ2. We have studied the predictors through three different experiments (based on the predictor sets defined in Table 7.4). Figure 7.2 shows the average AUCs of classification algorithms for all experiments. We expected to see a large difference in prediction performance among the three experiments. However, there is not much difference in prediction performance as we can see in Figure 7.2 the difference in average AUC is only ± 0.02. From this figure, we found out that the performances slightly degrade for experiment C, but the amount of degradation is not very significant. This means, even though the number of predictors in experiment C is smaller than in experiments A

Table 7.6: *Results for Predictor set C.*

No.	Project	J48	k-NN(1)	k-NN (5)	Function Logistic	Naïve Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
1	ArgoUML	0.50	0.69	0.69	0.54	0.56	0.50	0.55	0.50	0.64	0.60
		0.00	0.04	0.05	0.05	0.06	0.00	0.07	0.07	0.04	0.03
2	Mars	0.53	0.69	0.75	0.61	0.62	0.52	0.70	0.58	0.73	0.61
		0.06	0.03	0.05	0.05	0.13	0.05	0.07	0.16	0.09	0.08
3	JavaClient	0.76	0.83	0.86	0.81	0.79	0.78	0.75	0.80	0.86	0.81
		0.09	0.03	0.04	0.05	0.04	0.07	0.06	0.04	0.04	0.05
4	JGAP	0.54	0.60	0.62	0.67	0.66	0.51	0.59	0.65	0.72	0.60
		0.07	0.05	0.07	0.12	0.09	0.02	0.04	0.10	0.10	0.17
5	Neuroph	0.61	0.79	0.82	0.71	0.87	0.56	0.63	0.72	0.78	0.68
		0.14	0.06	0.06	0.10	0.04	0.09	0.08	0.16	0.09	0.08
6	JPMC	0.54	0.66	0.67	0.69	0.57	0.50	0.58	0.61	0.69	0.59
		0.08	0.06	0.06	0.08	0.08	0.01	0.03	0.06	0.09	0.08
7	Wro4j	0.63	0.70	0.68	0.77	0.77	0.62	0.70	0.69	0.74	0.68
		0.18	0.09	0.19	0.16	0.15	0.12	0.13	0.21	0.14	0.14
8	xUML	0.74	0.78	0.77	0.69	0.73	0.69	0.72	0.82	0.83	0.75
		0.06	0.07	0.06	0.12	0.06	0.10	0.06	0.04	0.06	0.06
9	Maze	0.67	0.61	0.64	0.60	0.68	0.58	0.63	0.60	0.70	0.59
		0.07	0.10	0.14	0.11	0.08	0.07	0.06	0.10	0.10	0.08
	No. of InfoGain ≥ 0.60	**5**	**8**	**9**	**7**	**7**	**3**	**6**	**6**	**9**	**5**
	Average	**0.61**	**0.71**	**0.72**	**0.68**	**0.69**	**0.58**	**0.65**	**0.66**	**0.74**	**0.66**
		0.09	**0.08**	**0.08**	**0.08**	**0.10**	**0.10**	**0.07**	**0.10**	**0.07**	**0.08**

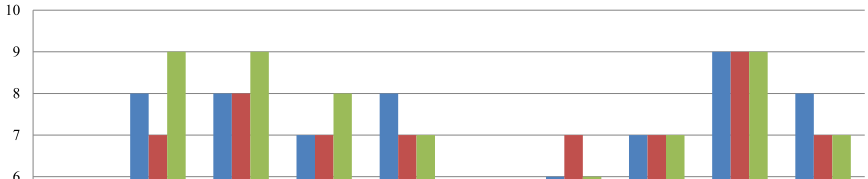
Note : The first row for each predictor set is the average AUC, the second row lists the standard deviation. Cells with AUC < 0.60 are highlighted.

and B, the set of predictors is still reliable for prediction purposes. This shows that the Coupling category (Predictor Set C) strongly influences the prediction performance.

7.5.2 Benchmark Scoring Results

RQ3. The classification algorithms were evaluated by measuring the average and standard deviation of the AUC over ten runs for each predictor set. Table 7.6 shows an example of results for experiment C. We have highlighted those cells that contain very weak classification results, i.e. AUC < 0.60. Note that an AUC of 0.50 means that the classifier produces completely random result. For our study, we consider a value of AUC of 0.60 or higher indicates a useful algorithm. This means, the classification algorithms that are able to produce this score for almost all case studies for all experiments are considered suitable for classifying key classes.

After performing the experiments, we found that the Random Forests and K-Nearest Neighbor (k-NN(5)) algorithms perform the best in classifying the key classes in class diagrams in terms of overall AUC, as well as robustness over various predictor sets. Figure 7.3 shows the prediction performance of all selected classification algorithm. This figure illustrates the number of case studies (for each predictor set) in which the



	J48	k-NN(1)	k-NN(5)	Function Logistic	Naïve Bayes	Decision Table	Decision Stump	RBF Network	Random Forest	Random Tree
Prediction Set A	5	8	8	7	8	3	6	7	9	8
Prediction Set B	5	7	8	7	7	3	7	7	9	7
Prediciton Set C	5	9	9	8	7	3	6	7	9	7

Figure 7.3: *AUC Score >= 0.60*

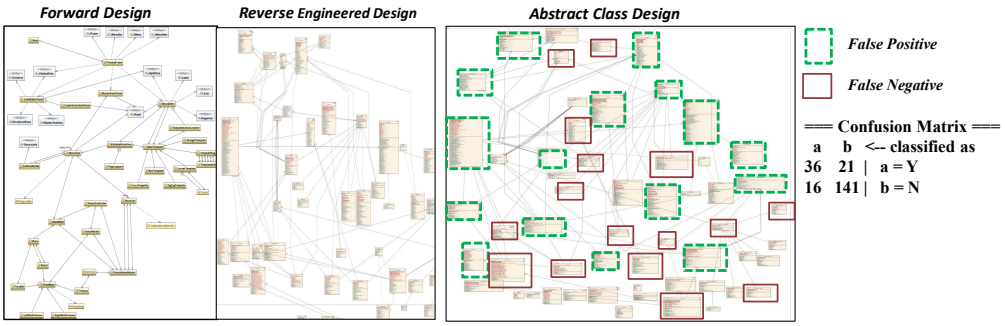


Figure 7.4: *Application of Random Forests Classification Algorithm.*

classification algorithm produces an AUC score greater than 0.60. Random Forests and k-NN(5) perform the best prediction where both classification algorithms produced AUC scores above 0.60 for at least 8 case studies of all datasets. Meanwhile, Naive Bayes, Random Tree, Function Logistic, RBF Network and Decision Stump performed less robust prediction across all predictor set. These classification algorithms performed reasonably well. They produced an AUC above the threshold for 6 to 8 case studies. J48 and Decision Table appear not to be suitable to be used in these case studies, given the low number of results with AUC $\geq$ 0.60 (between 3 to 5). The average AUC score of more than 0.72 for Random Forests and k-Nearest Neighbor (k-NN(5)) shows their suitability for all predictor set. Figure 7.4 illustrates the application of our method. In particular, it applies the Random Forests classification algorithm to the JavaClient case

study. As a result, a confusion matrix was generated. It shows that the total number of classes is 214 with 57 of the classes in the forward design. The generated confusion matrix shows that 36 out of 57 classes are correctly predicted as should be present in the class diagram. Also, 141 out of 157 classes are correctly predicted as should be omitted from the abstract class diagram. On the other hand, there are 21 false negatives (predicted as leave out, but should be included) and 16 classes that are false positives (predicted as 'include', but should not be included).

7.6 Discussion

With this result, we can conclude that the class diagram metrics from the Coupling and Size category can be good predictors for classifying key classes in class diagrams. In summary, there are three class diagram metrics that should be considered as influential predictors: Export Coupling Parameter (EC_Par), Dependency In (Dep_In) and Number of Operation (NumOps). This finding is consistent with the findings in Chapter 5 and Chapter 6 where the Number of Operation and Relationship (related to coupling) are the elements that are most software developers looked at in order to find the important classes in a class diagram.

The results show that k-NN(5) and Random Forests perform best and are suitable classification algorithms in this study. We took a step forward by exploring this classification algorithm by applying the algorithm individually to several case studies. As a result, some of the predicted True Positive in the algorithm k-NN(5) are predicted False Negative in the Random Forests and vice versa. We compared all the result manually from those two algorithms applied to several case studies and some of the true and false results are different. The possibility to enhance this predictive power is by combining those classification algorithms to achieve the best result. Given the imbalanced data, all selected algorithms were not able to produce high AUC scores.

This study was aimed at discovering suitable classification algorithms that could provide a rank score concordant with the likelihood for classes to be included in the UML class diagram. Based on this result, we are able to produce an approach for ranking classes for importance. This will allow the software engineer to generate a UML diagram at different levels of detail. To construct the abstraction of the class diagrams, the software engineer may apply the abstraction of relationship in class diagrams as presented by Egyed [52].

7.6.1 Threats to Validity

This study assumed that all the classes that existed in the forward designs were the important classes. There is a possibility that some of these classes were not important or not the key classes of the system. Also, there is a possibility that the forward design used is too 'old' or in other words obsolete compared to the version of the source code

used. Feedback from the system developer may enhance the accuracy of these key classes from forward design. However, collecting such feedback requires more effort.

The input of this study is dependent on the RE-CD constructed by the MagicDraw CASE tools. As mentioned in Chapter 4, there are several weaknesses of CASE tools' reverse engineering features. This weakness may influence the accuracy of the class diagram metrics calculation. There is a higher risk for large system that the CASE tool may leave out several information of some classes.

We only cover nine open source case studies. Based on the amount of classes, we can consider that the case studies represent small to medium size projects. The result may differ if we include large systems in our case studies.

7.7 Conclusion and Future Work

In this study, we proposed an approach for condensing RE-CD by selecting the key classes in it. We studied how well machine learning techniques perform in selecting the key classes in a class diagram by using supervised learning methods. The machine learning algorithms were trained on a set of open source projects. These projects contain a forward design class diagram which was used as a reference ('ground truth') for validating the quality of the condensation. Given the imbalanced nature of the data, Area under ROC curve was used as a performance evaluator for these algorithms.

This study evaluated (1) the influential predictors in classifying key classes and, (2) compared various machine learning classification algorithms on nine case studies derived from open source software projects, to identify candidate algorithms with the most accurate as well as robust behavior across predictor sets. We discovered that the Export Coupling Parameter, Dependency In and Number of Operation are the most influential predictors for classiying key classes in a class diagram. On these predictor sets, Random Forests and k-Nearest Neighbor provided the best results. For all listed case studies, the Random Forests method scores an AUC above 0.64 and the average AUCs for every prediction set is 0.74. These algorithms are able to produce a predictive score that can be used to rank important classes by relative importance. Based on this class-ranking information, a tool can be developed that provides views of RE-CDs at different levels of abstraction. In this was, developers may generate multiple levels of class diagram abstractions, ranging from highly detailed class diagram (equal to source code) to abstract class diagram (satisfying architect's preference for high-level views). In a broader perspective, this approach supports both the "Bottom-Up" and also the "Top-Down" approach for understanding of programs [157].

The results of this research may be improved by finding complementary explanatory variable. We also expect better results by taking the meaning of classes into account (see Chapter 8). Finding the set of projects suitable for this study was a very time-consuming task. This set can now be used by the scientific community as a benchmark for further studies.

7.7.1 Future Work

For future work, there is a number of ways to extend this work. Alternative input parameters for predicting the key classes in a class diagram could be investigated. This could include the use of other types of design metrics, for example, based on (semantics of) the names of classes, methods and predictors. There are also possibilities to use source code metrics such as Line of Code (LOC) and Lines of Comments as additional predictors for the classification algorithms. Moreover, we could look at the identification of 'features' as a unit of inclusion or exclusion in the UML class diagrams. Also, more extensive benchmarking should take place, for instance by learning models on one problem and testing it on another, or testing out an ensemble approach that combines classification algorithms. Specific approaches exist to better transfer knowledge across different problems, such as transfer learning.

Another approach to deal with limited availability of 'ground truth'-data for validation is to use a semi-supervised or interactive approach, where a user first selects some limited top level classes, then the system learns and recommends further classes to be included, and the user responds by confirming or rejecting the recommendations. Building an interactive application may also help to guide future research.

In terms of predictive performance, it could be interesting to compare the result of this study with other approaches. This study uses the classes in the forward design as the 'ground truth'. In version history mining, the classes that are frequently changing are seen as candidates for key classes [77]. It is also interesting to compare our approach with other works that apply different algorithms such as HITS web mining (used in [184]), network analysis on dependency graphs (used in [154]) and PageRank [136], and provide guidelines in which cases that approach would be preferred, or to create hybrid approaches.

We extend this research by validating the result of our proposed technique for condensing UML class diagrams in Chapter 10. The result of this study also allows us to create an automated tool to condense class diagrams. The automated tool is presented in Chapter 9 .

Chapter 8

Condensing Reverse Engineered Class Diagrams through Class Name Based Abstraction

In this chapter, we report on a machine learning approach to condensing class diagrams. This research focuses on building a classifier that is based on the names of classes in addition to design metrics, and we compare to earlier work that is based on design metrics only. We assess our condensation method by comparing our condensed class diagrams to class diagrams that were made during the original forward design. Our results show that combining text metrics with design metrics leads to modest improvements over using design metrics only. On average, the improvement reaches 5.3%. 7 out of 10 evaluated case studies show improvement ranges from 1% to 22%.

8.1 Introduction

In our previous work (Chapter 7), we proposed an approach to simplify RE-CDs based on static analysis. We used forward designs as 'ground truth' for what classes are most important, and then used machine learning techniques to learn the relationship between class characteristics measured in object-oriented design metrics, and the importance of the class (in forward design or not). The classifier model delivers a score

This chapter is adapted from a publication entitled **"Condensing Reverse Engineered Class Diagrams through Class Name Based Abstraction"**, 2014 World Congress on Information and Communication Technologies (WICT)

for each class that indicates the importance (likelihood to be in the forward design), so that developers can explore the class diagrams at different levels of abstraction by varying the threshold on the score.

In this chapter, we extend the previous work by introducing text based metrics derived from class names. From our previous survey (Chapter 5), we found that software engineers consider both design metrics (e.g. Coupling, Number of methods) and class element names (i.e. Related to domain) to be key features to decide on the importance of a particular class. The previous study (Chapter 7) is used as a baseline and we also follow the experimental setup for this research. We derive several text metrics from the set of class names. We then apply a set of classification algorithms to the text metrics and compare the result with the design metrics and a combination of both design and text metrics, using the Area Under the Curve (AUC) evaluation measure.

The contributions of this study are the following:

- Formulation of text metrics based on class names for classification of key classes.
- Combining text metrics and design metrics for classification of key classes.
- Evaluation of the approach shows that the combination of text and design metrics has improved the prediction performance.

This chapter is structured as follows: Section 8.2 discusses the related work. Section 8.3 describes the research questions. Section 8.4 explains the approach while Section 8.5 explains the experiment description. We present the analysis of results in Section 8.6 and discuss our findings in Section 8.7. This followed by conclusions and future work in Section 8.8.

8.2 Related Work

In this section, we discuss related work. We consider studies on the usage of text in software documentation and we found the research in the areas of code summarization and analysis of execution traces are related to our topic.

8.2.1 Code Summarization

Code summarization is an approach to summarize source code to help program comprehension. Haiduc et al. [74] introduced automated source code summarization based on the text retrieval approach. Their research used the natural language summarization technique to summarize the source code using lexical (i.e. identifiers and comments) and structural information (i.e. class, method, function). They generated a list of important keywords of methods. The generated important keywords were validated by six software developers. This preliminary study found that the method's names

influence the identification of important keywords in a method. Their study showed that 98.7% of identified keywords are derived from methods' name.

Haiduc et al. [75] extended the research by evaluating four text summarization techniques for the purpose of code summarization. For this evaluation, they generated various formations of text (e.g. by using weight – *tf-idf, binary-entropy, log*), as well as different types of the text summarization techniques. The generated keywords were validated by four software developers. They discovered that the combination of text summarization technique (i.e. Lead+Vector Space Model) performed the best result in capturing the meaning of methods and classes in an object-oriented source code. It is interesting to see that their research also indicates that class names are essential information in summarizing source code. This research has been replicated and extended by Eddy et al. [50]. Eddy et al. extended the work by evaluating a text retrieval technique (Hierarchical Pachinko Allocation Model). The validation was performed by 14 software developers, and their results confirmed Haiduc et al.'s findings.

Moreno et al. [114] proposed an automatic structured natural language summaries generator specifically for Java classes. Differently from the work by Haiduc et al., they use detailed structural class information (i.e. class stereotype) in addition to the class identifier to generate a summary of classes. Their validation illustrates that 90% of the generated class summaries were concise, readable and understandable. 69% of the summaries did not miss the important information.

Our study shares with text summarization that we look for cues of importance in the (key)words used in classes and method. Rather than aiming for a textual summarization, we target graphical summaries. A difference between our work and the text-based summarization approaches is that text-based summarization does not take information about class relations into account. From that perspective, our approach is more aimed at summarizing a design, while the text-based summarization is more focussed on summarization of individual classes (which may be used to summarize a system when seen as a set of classes). However, our work could benefit from mechanisms for selecting important information of classes for inclusion in class diagrams. We keep this as a proposal for future work.

8.2.2 Analysis of Execution Trace

Pirzadeh et al. [138] proposed a technique to simplify the analysis of execution traces for understanding a system's behavior. Their technique analyses execution traces by analyzing the most relevant information about the execution traces according to the execution phases.

Medini et al. [111] presented the Segment Concept AssigNer (SCAN) to assign labels to sequences of methods and to discover relations between segments by using execution traces as input. They utilized method-names, parameters and code-bodies. They applied text processing to filter the collected text.

Lin et al. [105] proposed Unsupervised Induction of Concepts (UNICON) to cluster large numbers of elements of semantically related words using an unsupervised technique. The inputs of their study are: (1) a collocation database and, (2) a similarity matrix of words. They used a clustering algorithm to break up a large set of data into small subsets. Then, they apply UNICON to suggest a set of concepts of the word.

The related works discussed above use text derived from the execution trace to understand the system. In contrast, we use the information that can be extracted from the source code and design documentation (static analysis). The volume of text derived from execution traces is so huge that the research in that line may only group the execution in phases. However, we acknowledge the use of information derived from execution traces as an interesting complement to our approach.

8.3 Research Questions

This section describes the main research question and sub-questions. The main question of this research is the following:

MQ: *Do text metrics derived from class names add value over design metrics to identify key classes to be included in a class diagram?*

In order to answer this question, the answers to the following questions need to be explored:

- **RQ1:** *What is the performance of text predictors?*
- **RQ2:**. *What are the most influential text predictor(s)?*
- **RQ3:**. *What is the performance of the classification algorithms in using text metrics as predictors?*
- **RQ4:**. *Which set of predictors produces the best result compared to design metrics?*

The research questions are answered in Section 8.6 and the main question is answered in Section 8.8.

8.4 Approach

In this section, we describe our approach in conducting this experiment. The overall framework of this experiment is shown in Figure 8.1. The input for this study are the system documents (*Step 1*); which consist of the RE-CD in XML Metadata Interchange (XMI) format and the Forward Design (FD). Section 8.4.1 describes the inputs in detail. Then, we normalize the text (class names) by removing the non-informational characters and words (*Step 2*). This includes converting class names into streams of words. In total, there are four subprocess involved, which are Lexical Analysis,

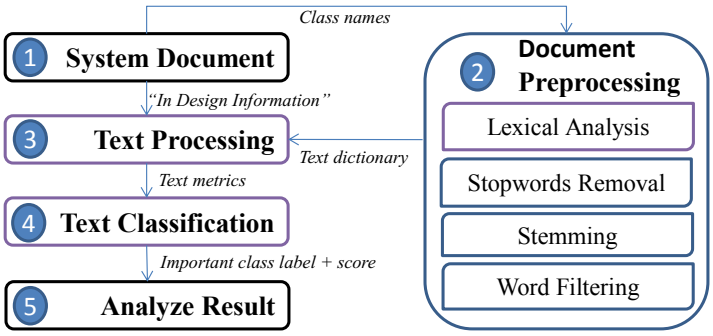


Figure 8.1: *Overall Framework*

Stopwords Removal, Stemming and Single Words Filtering. This process produces a text dictionary (Section 8.4.2 describes this process in detail). We then define the text metrics based on the text dictionary (*Step 3*). As a result, nine text metrics are invented. The detailed information about the text metrics is described in Section 8.4.3. For text classification (*Step 4*), we evaluate nine classification algorithms. Detailed information about this is described in Section 8.4.4. Finally, we analyse the important classes and classification score (*Step 5*).

8.4.1 System Document

A "raw" RE-CD is generated based on source code by using MagicDraw [9] (version 17.0). The class names in RE-CD will be used to form the predictors. Forward designs (FD) are the designs that were created during the development of the system. We assume all designs in the project's documentation are forward design. The machine learning algorithms use the FDs to learn the positive instances. The list of datasets (OSSD projects) is shown in Table 8.1. The datasets were collected from different types of domain. The detailed information about these projects can be found at Chapter 3.

8.4.2 Document Preprocessing

For data preparation, we use an automatic indexing procedure for English as mentioned in [175] and also the document preprocessing procedure suggested by [18]. This procedure includes lexical analysis, stop words removal, words filtering and stemming. The processes of this procedure are the following.

Lexical Analysis of the Text

Lexical analysis refers to a technique of converting a stream of characters into a stream of words [18]. One of the objectives of this technique is to identify the word from the class names collection. From our observation, most of the class names consisted of one

Table 8.1: *List of Case Study*

No.	Project	Total Classes in Re-CD **(S)**	Total Classes in FD **(D)**	D:S **ratio (%)**
1.	ArgoUML	903	44	4.87
2.	Mars	840	29	3.45
3.	JavaClient	214	57	26.64
4.	gwt-Portlets	178	20	11.24
5.	JGAP	171	18	10.52
6.	Neuroph	161	24	14.90
7.	JPMC	121	24	19.83
8.	Wro4J	87	11	12.64
9.	xUML	84	37	44.05
10.	Maze	59	28	47.45

or more words. Normally, capital letters are used to separate these words. Therefore, we separated the word(s) in class names from each document based on capital letters. The algorithm for class names separation is presented in Algorithm 1. For instance, the class name "PlayerSimulationData" will be separated into 3 different words: "Player", "Simulation" and "Data". There are also several exceptional cases where class names use multiple capital letters (such as "MMClassKeyword", "GAAlgorithm", "LMS" and "IACNeuron"). This issue is addressed by lines 16-17 in Algorithm 1. Then, the text documents generated from the lexical analysis process were loaded into a text processing tool. RapidMiner version 5.3 [11] was used as the text processing tool.

Stop Words Removal

This study requires only significant (meaningful) keywords. Stop words such as "the" or "and" help to build ideas, but do not carry any significance themselves [143]. Hence, these words are removed from the keywords' list. For example, the class name "PlantsAndAnimal" consists of the following words: "Plants", "And", and "Animal". The keywords "Plants" and "Animal" are meaningful keywords while the word "And" is only used to connecting those keywords. This process was done automatically by using the text processing tool (RapidMiner).

Stemming

A lot of words have the same root meaning but appear as different variants. A word for instance "maintenance" and "maintain" have the same word root. Hence, the stemming algorithm is used to resolve this issue. Stemming refers to a computational procedure that reduces all words with the same root (or, if prefixes are left untouched, the same stem) to a common form, usually by stripping each word of its derivational

Algorithm 1 Class Name Separation Algorithm

```
1:  Input :
2:  Document (D) = list of class names
3:  Output :
4:  Names = list of words
5:  Method :
6:  ClsNameChar = array of characters
7:  while D not End of List do
8:     Convert class names to array of characters (AC)
9:     for count = 0 to AC.size do
10:       if count = 0 and AC[count] is Uppercase then
11:          Add AC[count] to ClsNameChar {#Begin of a word}
12:       else if AC[count] is Uppercase and AC[count -1] is Lowercase then
13:          Add ClsNameChar to Names {#A word is completed}
14:          Empty ClsNameChar
15:          Add AC[count] to ClsNameChar {#Begin of new word}
16:       else if AC[count] is Uppercase and AC[count + 1] is UpperCase then
17:          Add AC[count] to ClsNameChar {#For words that use only capital letters}
18:       else
19:          Add AC[count] to ClsNameChar
20:       end if
21:    end for
22: end while
23: return Names
```

and inflectional suffixes [106]. For this, we used the Porter stemming algorithm [140] provided by the text processing tool (RapidMiner).

Word Filtering

Normally, a word consists of more than two characters. Hence, we remove all the single character words.

8.4.3 Text Processing

The results of the previous process allow us to create a text dictionary [122]). The text dictionary is formulated based on the following information:

- *NumKeyword*: Number of keyword(s) in a class name.
- *InDoc*: Number of documents (projects) where keywords occur.
- *TotalOcc*: The number of occurrences of words in all documents.
- *SpecOcc*: The number of occurrences of words in a specific document.

We formulated the text metrics based on the assumption that classes that have a lot of frequently used keywords most likely are candidate for classes that should be included (key classes) in the system. We make this assumption as we observe that many important classes indicated in the case studies documents consist of popular (or common) keywords. The list of common and uncommon words are illustrated in Table 8.2 and Table 8.3 respectively. The complete tables of common and uncommon words can be found at [123]. The formulation of the text metrics is the following:

$$K = \{k_1..k_n\} \text{ is a set of } n \text{ keywords} \tag{8.1}$$

$$F = \{f_1..f_m\} \text{ is a set of } m \text{ documents} \tag{8.2}$$

$$TotalOcc(k, f) = \text{ Number of Occurance of } k \text{ in } f \tag{8.3}$$

1. **NumKeyword** is the number of words in a class name. This metrics is gathered by simply calculating the number of word(s) in a class name after the lexical analysis process. The *NumKeyword* is represented as *K* in equation 8.1.

2. **ExiInDoc** counts the presence of a word in all documents. Therefore, *ExiInDoc* is defined as follows:

$$ExiInDoc(k, f) = \begin{cases} 1 & TotalOcc(k, f) > 0 \\ 0 & \text{Otherwise} \end{cases} \tag{8.4}$$

   Hence, we define presence function for a set of documents as:

$$ExiInDoc(k, F) = \sum_{f \in F} ExiInDoc(k, f) \tag{8.5}$$

   And therefore, we have:

$$ExiInDoc(K, F) = \sum_{k \in K} ExiInDoc(k, F) \tag{8.6}$$

   For instance, the class name "AdminConsoleDisplay" consists of: "Admin", "Console", and "Display". "Admin" appears in 3 documents, "Console" appears in 4 documents and "Display" appears in 6 documents. Hence, *ExiInDoc* for "AdminConsoleDisplay" is 13.

3. **MaxInDoc** takes the counts on the presence of a word; which is the highest count of all words in a class name. It is different from *ExiInDoc* where *MaxInDoc* compares the value of every keywords in a class name and takes the highest value. Hence, we define *MaxInDoc* as follows:

$$MaxInDoc(K, F) = \max_{k \in K} ExiInDoc(k, F) \tag{8.7}$$

   By using the example in *ExiInDoc*, the *MaxInDoc* value for "AdminConsoleDisplay" is 6.

4. **TotalOccAll** counts the total number of occurrences of each word in a class name from all documents. Therefore, the *TotalOccAll* for a class name is the following:

$$\mathrm{TotalOccAll}(K, F) = \sum_{k \in K} \mathrm{TotalOcc}(k, F) \tag{8.8}$$

   For example, the number of occurrences in all documents for "AdminConsoleDisplay" is 20 for "Admin", 12 for "Console" and 7 for "Display". Hence, the *TotalOccAll* for "AdminConsoleDisplay" is 39.

5. **MaxOccAll** takes the highest number of occurrences of a word (in all documents) in a class name.

$$\mathrm{MaxOccAll}(K, F) = \max_{k \in K} \mathrm{TotalOcc}(k, F) \tag{8.9}$$

   By using the example in *TotalOccAll* metric, the value of *MaxOccAll* for "AdminConsoleDisplay" is 20.

6. **TotalOccSpec** counts the total number of occurrences of each word in a class name from a specific document.

$$\mathrm{TotalOccSpec}(K, f) = \sum_{k \in K} \mathrm{TotalOcc}(k, f) \tag{8.10}$$

   Let say, the occurrences in a document for "AdminConsoleDisplay" is 10 for "Admin", 6 for "Console" and 2 for "Display". Hence, *TotalOccSpec* is 18.

7. **MaxOccSpec** takes the highest number of occurrences of a word in a class name from a specific document.

$$\mathrm{MaxOccSpec}(K, f) = \max_{k \in K} \mathrm{TotalOcc}(k, f) \tag{8.11}$$

   By using the example in *TotalOccSpec*, the *MaxOccSpec* is 10.

8. **WeightOccAll** is the average value of all word occurrences of a class name in all documents. It took the *TotalAccAll* and divided by the number of *NumKeyword* of a class name. Hence, the calculation *WeightOccAll* is the following:

$$\mathrm{WeightOccAll}(K, F) = \frac{\mathrm{TotalOcc}(K, F)}{\mathrm{NumKeyword}(K, F)} \tag{8.12}$$

9. **WeightOccSpec** is the average value of all word occurrences of a class name in a document. It took the *TotalOccSpec* and divided by the number of *NumKeyword*.

$$\mathrm{WeightOccSpec}(K, f) = \frac{\mathrm{TotalOcc}(K, f)}{\mathrm{NumKeyword}(K, f)} \tag{8.13}$$

Table 8.2: *The Top List of Common Words in Class Diagrams*

No.	word	inDoc	Total Occ	gwt-portlet	Argo UML	Java Client	JGAP	Mars	Neuroph	JPMC	xUML	Maze	Wro4J
1	panel	6	188	12	70	0	0	90	3	12	0	1	0
2	descriptor	2	171	0	1	0	0	170	0	0	0	0	0
3	player	2	153	0	0	152	0	1	0	0	0	0	0
4	model	7	139	0	97	0	0	20	1	6	4	7	4
5	uml	1	105	0	105	0	0	0	0	0	0	0	0
6	list	5	100	2	37	0	1	59	0	0	0	1	0
7	action	3	96	0	94	0	0	0	0	0	1	1	0
8	impl	4	84	7	5	0	0	1	0	0	71	0	0
9	resourc	4	75	0	33	0	0	37	0	1	0	0	4
10	prop	1	75	0	75	0	0	0	0	0	0	0	0
11	data	9	70	11	3	32	5	1	2	11	3	0	2
12	tabl	5	67	0	37	0	1	23	1	5	0	0	0
13	cr	1	67	0	67	0	0	0	0	0	0	0	0
14	tab	2	63	0	24	0	0	39	0	0	0	0	0
15	diagram	2	62	0	53	0	0	0	0	0	9	0	0
16	fig	1	61	0	61	0	0	0	0	0	0	0	0
17	state	4	59	0	50	3	0	0	0	0	5	1	0
18	type	6	55	0	15	0	1	31	4	0	3	0	1
19	config	5	54	1	3	21	8	21	0	0	0	0	0
20	posit	3	52	1	1	50	0	0	0	0	0	0	0
21	event	6	50	6	16	0	2	15	0	6	5	0	0
22	column	2	47	0	46	0	0	1	0	0	0	0	0
23	interfac	2	44	0	10	34	0	0	0	0	0	0	0
24	vehicl	1	43	0	0	0	0	43	0	0	0	0	0

Table 8.3: *The Top List of UnCommon Words in Class Diagrams*

No.	word	inDoc	Total Occ	gwt-portlet	Argo UML	Java Client	JGAP	Mars	Neuroph	JPMC	xUML	Maze	Wro4J
1	zoom	1	1	0	1	0	0	0	0	0	0	0	0
2	zig	1	1	0	0	0	0	0	0	0	0	1	0
3	zero	1	1	0	1	0	0	0	0	0	0	0	0
4	zag	1	1	0	0	0	0	0	0	0	0	1	0
5	yoga	1	1	0	0	0	0	1	0	0	0	0	0
6	xmi	1	1	0	1	0	0	0	0	0	0	0	0
7	write	1	1	0	0	1	0	0	0	0	0	0	0
8	wrapper	1	1	0	0	0	0	0	0	0	0	0	1
9	workout	1	1	0	0	0	0	1	0	0	0	0	0
10	wind	1	1	0	0	0	0	1	0	0	0	0	0
11	win	1	1	0	1	0	0	0	0	0	0	0	0
12	wi	1	1	0	0	1	0	0	0	0	0	0	0
13	weather	1	1	0	0	0	0	1	0	0	0	0	0
14	waypoint	1	1	0	0	1	0	0	0	0	0	0	0
15	wave	1	1	0	0	0	0	0	0	0	0	1	0
16	watch	1	1	0	0	0	0	0	0	0	0	0	1
17	wai	1	1	0	0	0	1	0	0	0	0	0	0
18	void	1	1	0	0	1	0	0	0	0	0	0	0
19	visual	1	1	0	0	0	1	0	0	0	0	0	0
20	violat	1	1	0	1	0	0	0	0	0	0	0	0
21	viewer	1	1	0	0	0	0	0	0	0	0	1	0
22	vi	1	1	0	1	0	0	0	0	0	0	0	0
23	vecmov	1	1	0	0	1	0	0	0	0	0	0	0
24	valid	1	1	0	0	0	1	0	0	0	0	0	0

8.4.4 Text Classification

The text classification process presents the classification activity to find the result of the class inclusion or exclusion in a class diagram. We use the Information Gain (InfoGain) Attribute Evaluator in WEKA [76] to estimate the predictive power of the text predictor. We prepared several sets of predictors. Then, we apply those sets of predictors to all selected classification algorithms to get the AUC score.

8.4.5 Analyze Result

The InfoGain measures and the AUC scores from the text classification process are analysed. We compare the performance of all evaluated classification algorithms across all datasets.

8.5 Experiment Description

This section explains the dataset that we used in this study and the evaluation measure for analyzing the results.

8.5.1 Dataset

All datasets in Chapter 7 were also used for this study. However, we added a project (gwt-Portlets) to the datasets that we found suitable for this study. The list of datasets is shown in Table 8.1. The datasets come from different types of domain. For instance, ArgoUML and xUML come from the UML tool domain, Neuroph comes from the neural network domain and JavaClient comes from the Application Programming Interface (API) domain. The detailed information about these projects can be found at Chapter 3.

The total number of classes in RE-CDs is between 59 and 903. The total number of classes in FDs is between 11 and 57. The ratio between FDs and RE-CDs (D:S ratio) is between 3% to 47%. For instance, let say if the D:S ratio of the JavaClient project is 26.6%, then only 26.6% of the classes that exist in the implementation (and hence in the RE-CD) are also in the FD. Meanwhile, another 73.4% of the classes that exist in the implementation does not appear in the FD. For the purpose of machine learning, only 26.6% of these classes are positive instances and the other 73.4% are negative instances.

With this information (Table 8.1), we see that most of these datasets are imbalanced in positive and negative instances.

8.5.2 Evaluation Measures

This subsection describes the evaluation methods that were used in this study. This evaluation is supported by the Waikato Environment Knowledge Analysis (WEKA)

[76]. The Information Gain Attribute Evaluator analysis [76] is used to evaluate the influence of each individual text metrics in predicting class inclusion/exclusion. This evaluation produces a score from 0 to 1 for every predictor. A value closer to 1 means strong influence.

We used the Area Under the Curve (AUC) [79] for analyzing the performance of the classification algorithm. WEKA provides the AUC calculation that produces a value ranging from 0 to 1. An AUC score approaching 1 means a better classification while a value close to 0.50 means the classifier performs almost randomly. The AUC score is used for evaluation because it may avoid the issue of favouring models that evaluation method just predict the majority outcome class. Because our data is typically imbalanced, such a majority-based score is suitable.

8.5.3 Experiment

This subsection describes the experiment set up in this study. We randomly split 50% the data in a 50% train and 50% test set. As mentioned before, the datasets are typically imbalanced. In the sense that, the classes that are in the RE-CD but not in the FD (the 'negatives') are highly dominated the entire dataset. If we use the 10-fold cross validation, the chance that many positives are included in the test data is very low because it uses 10% of the data for testing and 90% for training. Thus, the test set error measurements would not have been reliable. We ran each of the experiment 10 times using different randomizations to improve the reliability.

8.6 Analysis of Results

This section describes the analysis of results. Every subsection is presented to answer the questions specified in Section 8.3.

8.6.1 RQ1 : Influence of Predictors

We applied the Information Gain Attribute Evaluator analysis to evaluate the influences of individual predictors. The overall results of this evaluation are illustrated in Table 8.4. We consider a predictor as influential when the Information Gain value > 0. The column *Count* counts the number of times that a predictor produces InfoGain values > 0 (means influential) across all ten case studies. Overall, out of ten evaluated case studies, all individual text metrics are influential predictors for at least four case studies. This means all text metrics are capable of influencing the prediction of the important classes in a class diagram.

In terms of the predictive performance across case studies, the gwt-portlets, JPMC, Maze, Neuroph and Wro4j projects recorded poor performance (see Table 8.4). From our observation, the projects produce poor performance due to the fact that there is no

pattern in the class names in the positive instances group for most of the text metrics. Thus, the text metrics could not influence the prediction for these projects.

The predictive performance increases when a case study has several keywords (or *'champion words'*) that frequently appears in classes that grouped in the positive instances. For these classes, the text metrics produced relatively similar values that lead to homogeneity (or a uniform pattern) of data.

8.6.2 RQ2 : Most Influential Predictors

The *TotalOccSpec* and *MaxOccSpec* are the most influential predictors across all case studies. These predictors are influential in six case studies while the *WeightOccSpec* is influential in five case studies (Table 8.4). In addition, Table 8.4 also shows that the average InfoGain values of these three predictors are recorded as among the highest compared to other predictors. The top three most influential predictors come from the text derived from individual case studies. These top three predictors are highly correlated because these are derived from the same base information (*SpecOcc*). These predictors (related to text from a specific project) have higher predictive influence than metrics derived from all together.

8.6.3 RQ3 : Classification Algorithms Performance

In this study, we consider a classification algorithm to be suitable for prediction if its AUC score is above 0.60. Prior study (Chapter 7) shows that the suitable classification algorithm for class inclusion/exclusion based on design metrics are Random Forests and Nearest Neighbor. However, when using just text metrics, Logistic Regressions also provides suitable results. In addition, when design and text metrics are combined, Decision Stumps also perform surprisingly well (Table 8.5). For the xUML and JavaClient projects, AUC scores are higher than 0.60 for all evaluated classification algorithms.

8.6.4 RQ4 : Set of Predictors Performance

In this subsection, we present a comparison between the combination of text and design metrics with the text metrics and the design metrics separately. We also extend the analysis by comparing the sets of results by using the Random Forests classification algorithm that we found performed the best prediction.

Table 8.4: *Information Gain Results for Text Predictors*

No.	Text Metrics	Argo UML	gwt-portlet	Java Client	JGAP	JPMC	Mars	Maze	Neuroph	Wro4J	xUML	Count	Average
1.	TotalOccSpec	0.051	0.000	0.407	0.051	0.000	0.030	0.125	0.000	0.000	0.257	6	0.086
2.	MaxOccSpec	0.048	0.000	0.407	0.085	0.149	0.022	0.000	0.000	0.000	0.146	6	0.092
3.	WeightOccSpec	0.045	0.000	0.341	0.098	0.000	0.018	0.000	0.000	0.000	0.146	5	0.065
4.	TotalOccAll	0.038	0.000	0.512	0.000	0.000	0.030	0.000	0.000	0.000	0.109	4	0.032
5.	MaxOccAll	0.040	0.000	0.407	0.000	0.000	0.027	0.000	0.000	0.000	0.171	4	0.038
6.	WeightOccAll	0.033	0.000	0.259	0.000	0.000	0.025	0.000	0.000	0.110	0.000	4	0.025
7.	MaxInDoc	0.013	0.000	0.076	0.000	0.000	0.019	0.000	0.000	0.000	0.272	4	0.065
8.	NumKeyword	0.016	0.000	0.178	0.000	0.000	0.025	0.102	0.000	0.000	0.000	4	0.069
9.	ExiInDoc	0.013	0.000	0.152	0.000	0.000	0.021	0.000	0.064	0.000	0.000	4	0.043
	No. of InfoGain > 0	**9**	**0**	**9**	**3**	**1**	**9**	**2**	**1**	**1**	**6**		

Table 8.5: *Classification Algorithms Performance on Predictors Sets (AUC Score ≥ 0.60)*

Predictors sets	J48	k-NN (1)	k-NN (5)	Logi. Regr	Naive Bayes	Dec. table	Dec. Stump	RBF	Rand. Forest	Rand. Tree
Text (T)	4	5	9	9	8	2	6	7	9	5
Design (D)	5	8	9	7	8	3	7	7	9	6
Design_Text (DT)	7	6	9	8	9	4	9	9	9	6
Δ(T-D)*	-1	-3	0	2	0	-1	-1	0	0	-1
Δ(DT - T)*	3	1	0	-1	1	2	3	2	0	1
Δ(DT - D)*	2	-2	0	1	1	1	2	2	0	0

** Note : Positive values indicate improvement and negative values indicate degradation*

Combination of Text and Design Metrics vs. Text Metrics

In Table 8.5, Δ(T-D) shows the classification performance comparisons between the text metrics and the design metrics as predictors. The results in Table 8.5 show the ability of classifiers to produce AUC scores ≥ 0.60 to all case studies. It shows that the performance of J48, k-NN(1), Decision Table, Decision Stump and Random Tree degrades while k-NN(5), Naive Bayes, RBF and Random Forests do not show any changes in performance. Only Logistic Regression shows an improvement where it could predict 2 more projects (compared to design metrics). We take a step further by combining the text metrics and the design metrics. We compare the performance of the combination of these metrics with the individual text metrics and design metrics performance. The comparison between the combination of design and text metrics (DT) and the text metrics (T) is shown in Δ(DT-T). Seven classification algorithms show improvement when using a combination of design and text metrics while k-NN(5) and Random Forests perform the same. Only logistic regression performance degrades but the difference is only one project.

Combination of Design and Text Metrics vs. Design Metrics

In Table 8.5, Δ(DT-D) shows that using a combination of text and design metrics performs better than using only design metrics. Six classification algorithms show improvements while three classification algorithms present the same result for both sets of predictors. However, k-NN(1) shows degradation in classification performance for Δ(T-D) and Δ(DT-D).

Random Forests in Detail

We analyse the results of the Random Forests algorithm in detail by comparing the prediction score for the three evaluated predictor sets (D, T and DT). In Table 8.6, five projects demonstrate that the text metrics produce better results than the design metrics. The improvement of text metrics using the Random Forests classifier is between 2% to 7%. However, another five projects show degradation. The degradation recorded are between -13% to -21%. On average, out of ten projects, the text metrics show degradation of 5.6% compared to the design metrics. From our observation, the text metrics perform better prediction than the object-oriented design metrics when the total InfoGain value of text metrics is higher than the total InfoGain value of design metrics (in other words, the text metrics dominate the influential features).

Table 8.6 presents that prediction performance by using a combination of both metrics is better than using only the design metrics. Seven out of ten projects show positive improvement. The improvement is between 1% to 22%. On average, by using the combination of both metrics leads to an improvement of 5.3% over just design metrics. Based on this data, it seems that the combination of the design and text metrics produces the best result for the prediction of class inclusion/exclusion.

Table 8.6: *Random Forests Result for Predictors Sets*

Project	Text (T)	Design (D)	Δ (T-D)	Improv. (%)	Design + Text (DT)	Δ (DT-D)	Improv. (%)
ArgoUML	0.672	0.655	0.017	2.60	0.799	0.144	21.98
gwt-Portlets	0.603	0.564	0.039	6.91	0.597	0.033	5.85
JavaClient	0.895	0.844	0.051	6.04	0.929	0.085	10.07
JGAP	0.788	0.748	0.040	5.35	0.794	0.046	6.15
JPMC	0.733	0.692	0.041	5.92	0.773	0.081	11.71
Mars Simulation	0.645	0.766	-0.121	-15.80	0.830	0.064	8.36
Maze	0.584	0.674	-0.090	-13.35	0.635	-0.039	-5.79
Neuroph	0.683	0.835	-0.152	-18.20	0.849	0.014	1.68
Wro4J	0.588	0.742	-0.154	-20.75	0.695	-0.047	-6.33
xUML	0.727	0.847	-0.120	-14.17	0.840	-0.007	-0.83
Average	**0.692**	**0.737**	**-0.045**	**-5.55**	**0.774**	**0.037**	**5.28**

8.7 Discussion

In this section, we summarize and explain the result in the previous section. We also describe the threat to validity for this study.

8.7.1 Text Metrics Predictors Performance

From the results in Section 8.6.1, we know that all individual text metrics are influential predictors. However, Table 8.4 shows that the text metrics are suitable for only four case studies (i.e. ArgoUML, JavaClient, Mars and xUML) The text metrics were not able to influence the prediction of the gwt-portlets. From our further analysis, the design metrics also were not influential in this case study. For this case study, the text metrics produce a better result where it can produce the AUC ≥ 0.60 (as shown in Table 8.7). This is not possible to be achieved using design metrics predictor. Probably, the combinations of the text metrics and design metrics for gwt-Portlets project influence the prediction. The InfoGain is suitable to estimate the individual predictive power. We could not see the influence of the combination of the text metrics and design metrics through InfoGain measure.

8.7.2 Classification Algorithms

By using the text metrics, we found that the Random Forests, k-NN and Logistic Regression are suitable to be used for prediction. However, by using a combination of the design and text metrics as predictors, we found that the k-NN, Naive Bayes, Decision Stump, Radial Basis Function (RBF) and Random Forests are suitable for this purpose. From these algorithms, the Random Forests classifier was chosen for the validation experiment because it produced the best AUC score among those classifiers. From the results, we found that the combination of the design and text metrics is the best set of predictors. Only k-NN (1) classifier showed a decrease in performance. The performance of the text metrics predictors for this classifier maybe influences this result.

8.7.3 Application of Classification Method

Based on the previous section, we have found that Random Forests is the most suitable classification algorithm for our approach. The Random Forests classifier is capable of producing scores for every class in a class diagram. Based on this, we developed a tool (called Software Architecture Abstractor (SAAbs) [124]) to apply our approach to the RE-CD (see Chapter 9). The tool produces a score of importance for all classes in a class diagram. A higher score indicates that the class is more important or could be included in class diagrams. With this ranking, abstractions of a class diagram based on the importance of classes can be constructed.

Table 8.7: *Classification Result from Text Predictor (T)*

Project	J48	k-NN (1)	k-NN (5)	Logi. Regr.	Naive Bayes	Dec. table	Dec. Stump	RBF	Rand. Forest	Rand. Tree	Project Performance
ArgoUML	0.54	0.59	0.66	0.77	0.80	0.53	0.75	0.76	0.69	0.56	**6**
std. dev	*0.08*	*0.06*	*0.08*	*0.04*	*0.02*	*0.09*	*0.06*	*0.02*	*0.07*	*0.04*	
gwt_portlet	0.52	0.60	0.61	0.65	0.63	0.50	0.57	0.53	0.63	0.56	**5**
std. dev	*0.05*	*0.07*	*0.10*	*0.13*	*0.10*	*0.00*	*0.06*	*0.10*	*0.12*	*0.10*	
JavaClient	0.84	0.85	0.90	0.86	0.82	0.80	0.72	0.86	0.90	0.82	**10**
std. dev	*0.04*	*0.04*	*0.03*	*0.05*	*0.03*	*0.06*	*0.04*	*0.04*	*0.04*	*0.07*	
JGAP	0.54	0.62	0.73	0.78	0.78	0.50	0.71	0.76	0.78	0.60	**8**
std. dev	*0.08*	*0.07*	*0.07*	*0.08*	*0.05*	*0.00*	*0.03*	*0.06*	*0.07*	*0.10*	
JPMC	0.74	0.61	0.66	0.70	0.64	0.52	0.72	0.68	0.73	0.61	**9**
std. dev	*0.05*	*0.07*	*0.07*	*0.07*	*0.07*	*0.06*	*0.02*	*0.11*	*0.12*	*0.09*	
Mars-Simulation	0.50	0.56	0.69	0.78	0.77	0.50	0.67	0.81	0.64	0.59	**6**
std. dev	*0.00*	*0.08*	*0.05*	*0.03*	*0.02*	*0.00*	*0.03*	*0.02*	*0.07*	*0.05*	
Maze	0.60	0.56	0.51	0.57	0.48	0.51	0.54	0.50	0.60	0.57	**2**
std. dev	*0.09*	*0.08*	*0.08*	*0.11*	*0.09*	*0.02*	*0.04*	*0.06*	*0.07*	*0.08*	
Neuroph	0.55	0.59	0.69	0.71	0.65	0.50	0.57	0.62	0.68	0.61	**6**
std. dev	*0.05*	*0.06*	*0.06*	*0.12*	*0.06*	*0.00*	*0.06*	*0.06*	*0.08*	*0.07*	
Wro4j	0.50	0.57	0.65	0.70	0.50	0.50	0.53	0.47	0.57	0.59	**2**
std. dev	*0.05*	*0.06*	*0.12*	*0.21*	*0.19*	*0.01*	*0.09*	*0.20*	*0.15*	*0.11*	
xUML	0.68	0.64	0.64	0.77	0.72	0.62	0.60	0.73	0.73	0.63	**10**
std. dev	*0.08*	*0.10*	*0.05*	*0.08*	*0.07*	*0.08*	*0.03*	*0.06*	*0.04*	*0.06*	
Algorithm Performance	**4**	**5**	**9**	**9**	**8**	**2**	**6**	**7**	**9**	**5**	

8.7.4 Threats to Validity

This subsection describes the threats to validity of this study.

Threats to Internal Validity. The splitting of class names is done by separating the words based on the occurrence of the capital letters. However, not all designs follow this naming convention. There are several uses of characters such as "XYZ" and "123" that do not carry any significant meaning. This leads to inaccuracy of counting the occurrences of words.

Threats to External Validity. It is difficult to find open source software systems that have both FD and source code. Hence, we use ten open source software systems in our evaluation. These systems fall into the class of small to medium size software projects. Hence, we could not generalize the result of this study of all software systems.

Threats to Construct Validity. To measure the classification algorithm performance, we use the AUC as our evaluation metrics. The AUC can be considered as a standard metrics in data mining [78] which is designed to evaluate imbalanced datasets [141]. Thus, we believe there is little threat to construct validity.

8.8 Conclusion and Future Work

This chapter aimed to come up with better methods for abstracting class diagrams out of source code. To this end, we focused on the use of class names as a way of improving the prediction of class inclusion/exclusion. In our study, we introduced text metrics that capture the frequency of occurrence of class names. We found text metrics may help to improve the results. However, using the text metrics in isolation does not produce better predictions. A combination of text metrics and design metrics (DT) produces the best result. We also studied which classifier algorithm works best for abstracting classes. Out of nine evaluated classification algorithms, all evaluated classification algorithms showed improvement in classifying class inclusion/exclusion by using the combination of text and design metrics except for the k-NN(1) algorithm. The evaluation was done by comparing the classification result with the result produced by using design metrics as predictors. The Random forest classifier is the most suitable for our study. By using this classification algorithm, we may improve the prediction on average by 5.3%; the improvement for different software projects ranges between -6% to 22%.

For future work, we plan to use additional sources of text in the source code (the methods, parameter and attribute names) to enhance the text metrics. We also conduct a user study on the usage of resultant diagrams constructed by the tool, which are developed using this approach (see Chapter 10).

Chapter 9

Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams

A large fraction of the time consumed in software development and maintenance is spent on understanding the software, which indicates it is a critical activity. Software documentation, including software architecture design documentation, is an important aid in software comprehension. However, keeping documentation up-to-date with evolving source code is often challenging and absence of an update or more comprehensive design-level documentation is not uncommon. As a solution, software architecture design may be recovered using reverse engineering process. However, existing methods in the reverse engineering process produce complete design diagrams that include all the details that exist in the source code. The absence of abstraction from implementation details limits the usefulness of reverse engineering process for understanding software.

This research aims to address this problem by providing a method and a tool that allows developers to interactively explore a reverse engineered class diagram at scalable levels of abstraction. To this end, we propose a Software Architecture Abstraction (SAAbs) framework and an automated tool that implements this framework. The SAAbs framework applies a machine learning scoring algorithm to produce a class importance ranking for class diagrams; this ranking is the basis for software architecture abstraction and visualization.

Part of this chapter is adapted from a publication entitled **"Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams"**, In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014)

9.1 Introduction

Software comprehension is one of the most crucial tasks in software maintenance. It consumes a lot of time and effort, especially for large and complex systems. The documentation of software architecture or design is a highly useful artifact for system comprehension. However, it is common that this artifact is not kept up-to-date. Reverse engineering is one of the best options for recovering software architecture design information from the implementation code. Current reverse engineering methods suffer from several problems; one of them is that these methods reverse engineer the complete design, without focusing on what is more important, resulting into diagrams with too much information. Recent Computer Aided Software Engineering (CASE) tools offer to leave out several properties in a class diagram such as attributes and parameters. However, these tools do not identify which classes are more and which are less important, making it hard for the developer to focus or quickly understand the design at various levels of abstraction. A study by Fernandez-Saez et al.[59] found that many subjects in their controlled experiment did not consider reverse engineered diagrams to be helpful in maintaining software, which might be caused by the information overload in these class diagrams.

Often, when a new software developer is assigned to a maintenance task, several common questions may arise as part of the software comprehension activity. For instance, Where to start? Which classes are important? As software documentation is commonly out-of-sync, the software comprehension task becomes more difficult. Therefore, a software exploration tool is needed that fulfills several cognitive elements. Storey et al. [156] identify a number of cognitive elements important in software comprehension, we focus on the following: **E3**: Provide Abstraction Mechanisms; **E5**: Provide overviews of the system architecture at various levels of abstraction; **E4**: Support goal-directed hypothesis-driven comprehension; **E6**: Support the construction of multiple mental models; **E11**: Indicate the maintainer's current focus; and **E15**: Provide effective presentation styles.

In our previous chapters (7 and 8), we have conducted two studies that leverage static software design metrics to predict the relative importance of classes in class diagrams. These chapters study the usage of object-oriented metrics and class element names as features. The results suggest that the combination of both sets of features produces the best prediction (compared to the individual set of features), with Random Forests providing the most accurate and robust prediction results. Our proposed framework builds on those findings. For a more in depth review of the machine learning aspects, we refer to these chapters.

The main goal of this research is to provide an overview of the framework that can be used to develop an automatic tool to assist software comprehension through highlighted UML class diagrams. For this purpose, we introduce the Software Architecture Abstraction (SAAbs) framework. The SAAbs framework applies machine learning

classification algorithm to produce an ordered list of classes based on predicted importance. The list is then used by the SAAbs tool to generate UML class diagrams of various levels of abstraction.

The contributions of this chapter are the following:

- A tool-supported method for Software Architecture Abstraction using UML Class Diagrams;
- Multi-level abstraction and multi-level of detail visualization of Software Architecture through UML class diagram.

This chapter is structured as follows: Section 9.2 discusses the related research and Section 9.3 describes the SAAbs Overview. Section 9.4 discusses the tool. This followed by the conclusions and future work in Section 9.5.

9.2 Related Work

The core capability of the SAAbs framework is to produce a score that reflects the relative importance of classes in a reverse engineered complete class diagram. Therefore, we found the following studies are related.

9.2.1 The Usage of Network Metrics

Steidl et al. [154] perform network analysis on dependency graphs to identify important classes in a system. An empirical study was conducted to find the best combination of these network analysis metrics for class importance prediction. The prediction performance was validated by comparing prediction results with the classes recommended by project developers.

Thung et al. [164] extends the work by Osman et al. [127] by analyzing network metrics as features for predicting what classes in a system are important. They compared the results with the object-oriented design metrics in [127]. Their study discovered that the prediction performance of Random Forests is better after they used network metrics compared to object-oriented metrics. They found that the object-oriented metrics are reliable features for prediction.

9.2.2 The Usage of Software Version History

Hammad et al. [77] propose an approach that assigns importance scores to classes and sets of collaborating classes based on the frequency of changes of classes in software evolution. The importance scores are assigned based on the number of commits made for a class in a version control system. Itemset mining was used for assigning scores to sets of related classes.

Bieman et al. [24] investigate a method to identify and visualize frequently changed classes. The analysis of their study involves the implementation structure of the system and classes change logs. They present measures for class change-proneness i.e. Local change-proneness, Pair change coupling and Sum of pair coupling.

9.2.3 Other Related Work

The Software Model Extractor (SoMoX) is a software analysis tool that capable to reverse engineer a source code into component models. The SoMoX tool uses abstract syntax tree (AST) as the input data model to detect components based on hierarchical clustering of basic components derived from classes and interface [90].

Mancoridis et al. [108] investigate a technique to produce a high-level system organization of source code by using automatic clustering. They propose an automated software modularization environment to construct a hierarchical view of the system organization based on source code. The software extracts the module-level dependency from source code, cluster the entities and visualize the output.

In contrast to the works mentioned above, our proposed framework identifies the important class diagram by combining the object-oriented design metrics and text metrics as the features. The Random Forests classification algorithm is used as we found performing the best in Chapter 7 and 8.

9.3 SAAbs Overview

This section explains the overall framework of SAAbs and the implementation of each step in the framework to the SAAbs tool.

The overall framework of SAAbs is shown in Figure 9.1. This framework consists of three stages: Input, Process and Output. In the Input stage, the XMI file format of the system-to-analyze design source is identified (step 1, see Section 9.3.1). In the Process stage, the XMI file is parsed to retrieve the system-to-analyze design structure (i.e., the classes, attributes, relationships, textual information; step 2, Section 9.3.2). We derive two types of features from the design structure: Object-Oriented Design Metrics and Text metrics (Section 9.3.3). In addition, the user establishes 'ground truth' by loading a forward design, applying a given set of heuristics, or identifying a small set of key classes manually (step 4, Section 9.3.3). The classification process then learns a prediction model, and applies it to all classes to produce a score that reflects the relative importance of the class (detail in Section 9.3.4). This information allows the classes to be ranked based on its importance in the class diagram (step 5, Section 9.3.5). This ranking of classes is used to create various types of visualization for architecture abstraction of the system-to-analyze (step 6, 9.3.6). The implementation of the SAAbs tool is described in Section 9.3.7.

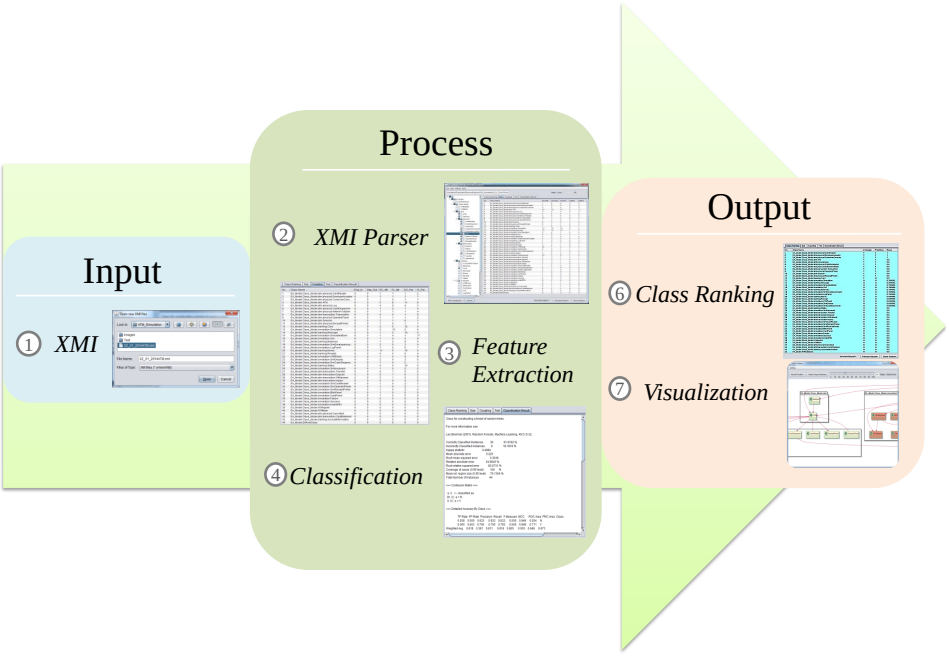


Figure 9.1: *Overall Framework : Input, Process and Output*

9.3.1 Input: XMI

An XMI file is the primary input for this tool. XMI is an XML-variant that is purposely invented to enable the exchange of UML models metadata information. However, the role of XMI file is not yet uniform for software design because every CASE tool generates their own flavour of XMI file even though they used the same structure. This issue still not resolved even though it has already been mentioned in 2003 by Stevens [155]. Our tool supports XMI versions 1.0, 1.1, 1.2, 2.0 and 2.1 as an input. It also allows multi-flavor of XMI and multiple version of XMI.

9.3.2 Process: XMI Parser

The XMI parser extracts all relevant information from the class diagram. We use the SDMetrics [180] OpenCore library that provides a robust XMI parser to solve the multi-flavor XMI issue. This parser has been proven in Chapter 4 to extract software metrics out of XMI files generated by eight CASE tools. Using this parser, we extract the classes, operations, attributes and relationship information.

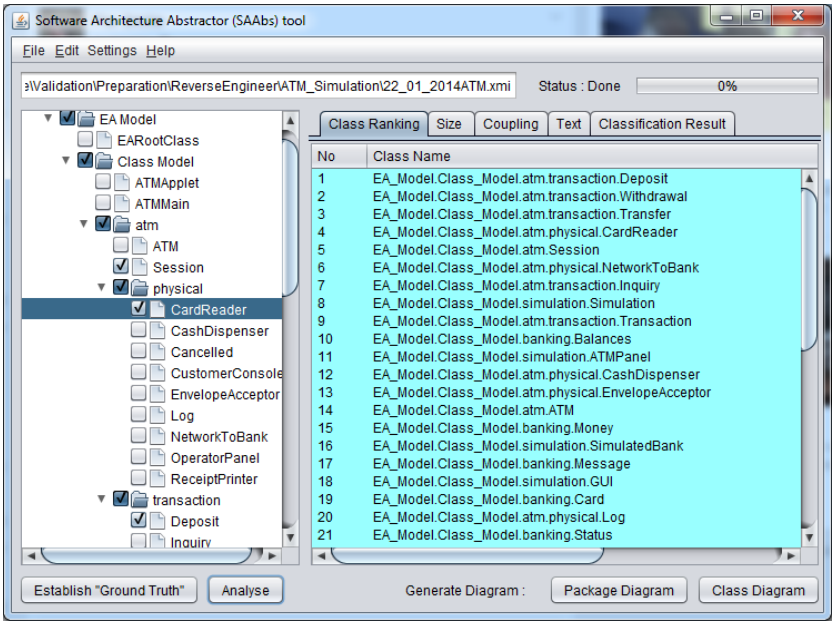


Figure 9.2: *Selection of the Candidate-Important Classes*

9.3.3 Process: Feature Extraction

The main purpose of this step is to prepare the data for the classification process, which includes selecting a number of key classes as 'ground truth' to be used for learning, and creating the low-level required predictor features from the class information extracted in the previous step (i.e. Object-oriented design metrics and Text metrics).

To create and validate a model that can rank all classes on relative importance, 'ground truth' information is required that identifies a number of key classes. The result in Chapter 7 demonstrated that candidates for important classes can be derived from the system's forward designs. Hence, we allow the user to load a forward design, or identify some key classes manually based on domain knowledge. Figure 9.2 shows the example of important classes candidates selection.

Alternatively, a heuristical algorithm can be used to set 'ground truth'. The heuristical algorithm is based on prior research that indicated that classes that have high coupling and high number of operations, are more likely to be the important classes in a class diagram ([132] and [133]). Thus, we can use this information to rank candidate-classes automatically as 'ground truth'. Algorithm 2 shows the algorithm for "Establish Ground Truth". Users can indicate the percentage of candidate classes that will be used for learning. For example, if a user indicates 25%, the tool selects the top 25% classes of the ranked list as the true positive instances. The 75% of classes that were not selected become the true negative instances for the machine learning.

Algorithm 2 "Establish Ground Truth" Rank Algorithm.

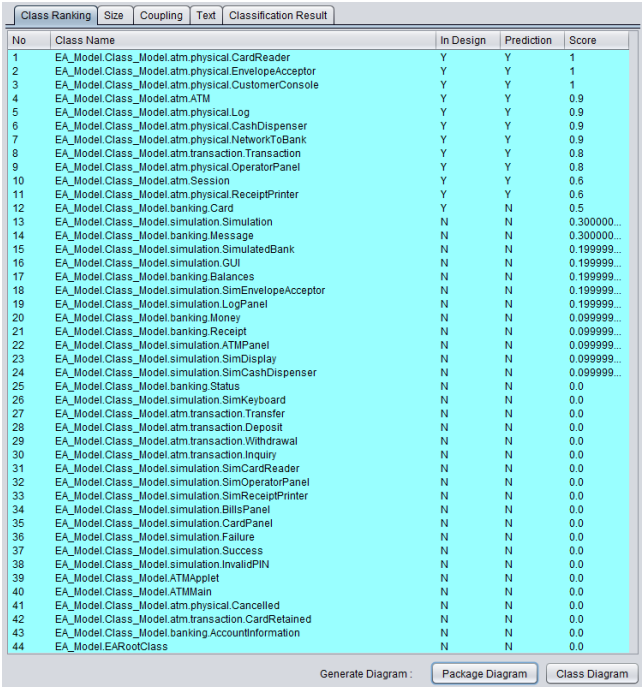
```
1: Input :
2: Document (D) = Design Document (XMI)
3: Output :
4: ListImpClsCand = Table of Class Metrics
5: Method :
6: Dep_In = Dependency In
7: Dep_Out = Dependency Out
8: EC_Par = Export Coupling Parameter
9: EC_Attr = Export Coupling Attribute
10: IC_Par = Import Coupling Parameter
11: IC_Attr = Import Coupling Attribute
12: NumOps = Number of Operation
13: CouplingMeasure = Sum of Coupling Metrics
14: Extract all Metrics to ListImpClsCand
15: add Column SumCoupling to ListImpClsCand
16: while D not EOF do
17:    CouplingMeasure = 0
18:    CouplingMeasure = Sum of (Dep_In, Dep_Out, EC_Par, EC_Attr, IC_Par, IC_Attr)
19:    add CouplingMeasure to SumCoupling Column
20: end while
21: sort ListImpClsCand based on SumCoupling, NumOps
22: remove SumCoupling Column
23: return ListImpClsCand {return ranked classes metrics}
```

One may expect that the actual prediction result based on this "ground truth" would be the same as the one automatically suggested by this Establish Ground Truth algorithm. However, this is not the case: the actual classifier suggests other important classes instead of the ones provided as "ground truth". One of the reasons is that the machine learning algorithm uses more fine-grained features than the ones used to suggest the "ground truth". That said, this method should be seen more as a backup alternative to using the forward design or manual selection of key classes, which should be seen as a more appropriate way of setting ground truth. The heuristic algorithm may actually support a user in carrying out this task by suggesting classes to consider.

After 'ground truth' has been set, relevant low-level features are extracted from available design information. From Chapter 7 and 8, we concluded that the combination of Object-oriented Metrics and Text Metrics is capable of producing a good prediction of important classes. We list the object-oriented metrics to be used according to Chapter 7 as items 1-11 in Table 9.1. Item number 11 to 20 in Table 9.1 are the text metrics that are formulated from the class names.

Table 9.1: *The List of Prediction Features*

No	Metrics	Description
1	NumAttr	The number of attributes of a class.
2	NumOps	The number of operations of a class (Weighted Methods per Class (WMC) in [40] and Number of Methods (NM) in [99]).
3	NumPubOps	The number of public operations of a class (also known as Number of Public Methods (NPM) in [99]).
4	Setters	The number of operations with the operation names starting with 'set'.
5	Getters	The number of operations with the operation names starting with 'get', 'is', or 'has'.
6	Dep_Out	The number of dependencies in which the class is the client.
7	Dep_In	The number of dependencies in which the class is the supplier.
8	EC_Attr	The number of times the class is externally used as attributes type (a version of OAEC +AAEC in [34]).
9	IC_Attr	The number of attributes in a class that has another class or interface as their type (a version of OAIC+AAIC in [34]).
10	EC_Par	The number of times the class is externally used as parameter type (a version of OMEC+AMEC in [34]).
11	IC_Par	The number of parameters in the class that has another class or interface as their type (a version of OMIC+AMIC in [34]).
12	NumKeyword	The number of words contains in a class name [128].
13	ExiInDoc	Counts the presence of a word in all documents [128].
14	MaxInDoc	The highest number of presence of a word from a class name in all documents [128].
15	TotalOccAll	Counts the total number of occurrences of each word in a class name from all documents [128]
16	MaxOccAll	The highest number of occurrences in all documents of a word in a class name [128].
17	TotalOccSpec	Counts the total number of occurrences of each word in a class diagram in a specific document [128].
18	MaxOccSpec	The highest number of occurrences of the word in a class name in a specific document [128].
19	WeightOccAll	The average value of all word occurrences of a class name in all documents [128].
20	WeightOccSpec	The average value of all word occurrences of a class name in a document [128].



Class Ranking | Size | Coupling | Text | Classification Result

No	Class Name	In Design	Prediction	Score
1	EA_Model.Class_Model.atm.physical.CardReader	Y	Y	1
2	EA_Model.Class_Model.atm.physical.EnvelopeAcceptor	Y	Y	1
3	EA_Model.Class_Model.atm.physical.CustomerConsole	Y	Y	1
4	EA_Model.Class_Model.atm.ATM	Y	Y	0.9
5	EA_Model.Class_Model.atm.physical.Log	Y	Y	0.9
6	EA_Model.Class_Model.atm.physical.CashDispenser	Y	Y	0.9
7	EA_Model.Class_Model.atm.physical.NetworkToBank	Y	Y	0.9
8	EA_Model.Class_Model.atm.transaction.Transaction	Y	Y	0.8
9	EA_Model.Class_Model.atm.physical.OperatorPanel	Y	Y	0.8
10	EA_Model.Class_Model.atm.Session	Y	Y	0.6
11	EA_Model.Class_Model.atm.physical.ReceiptPrinter	Y	Y	0.6
12	EA_Model.Class_Model.banking.Card	Y	N	0.5
13	EA_Model.Class_Model.simulation.Simulation	N	N	0.300000...
14	EA_Model.Class_Model.banking.Message	N	N	0.300000...
15	EA_Model.Class_Model.simulation.SimulatedBank	N	N	0.199999...
16	EA_Model.Class_Model.simulation.GUI	N	N	0.199999...
17	EA_Model.Class_Model.banking.Balances	N	N	0.199999...
18	EA_Model.Class_Model.simulation.SimEnvelopeAcceptor	N	N	0.199999...
19	EA_Model.Class_Model.simulation.LogPanel	N	N	0.199999...
20	EA_Model.Class_Model.banking.Money	N	N	0.099999...
21	EA_Model.Class_Model.banking.Receipt	N	N	0.099999...
22	EA_Model.Class_Model.simulation.ATMPanel	N	N	0.099999...
23	EA_Model.Class_Model.simulation.SimDisplay	N	N	0.099999...
24	EA_Model.Class_Model.simulation.SimCashDispenser	N	N	0.099999...
25	EA_Model.Class_Model.banking.Status	N	N	0.0
26	EA_Model.Class_Model.simulation.SimKeyboard	N	N	0.0
27	EA_Model.Class_Model.atm.transaction.Transfer	N	N	0.0
28	EA_Model.Class_Model.atm.transaction.Deposit	N	N	0.0
29	EA_Model.Class_Model.atm.transaction.Withdrawal	N	N	0.0
30	EA_Model.Class_Model.atm.transaction.Inquiry	N	N	0.0
31	EA_Model.Class_Model.simulation.SimCardReader	N	N	0.0
32	EA_Model.Class_Model.simulation.SimOperatorPanel	N	N	0.0
33	EA_Model.Class_Model.simulation.SimReceiptPrinter	N	N	0.0
34	EA_Model.Class_Model.simulation.BillsPanel	N	N	0.0
35	EA_Model.Class_Model.simulation.CardPanel	N	N	0.0
36	EA_Model.Class_Model.simulation.Failure	N	N	0.0
37	EA_Model.Class_Model.simulation.Success	N	N	0.0
38	EA_Model.Class_Model.simulation.InvalidPIN	N	N	0.0
39	EA_Model.Class_Model.ATMApplet	N	N	0.0
40	EA_Model.Class_Model.ATMMain	N	N	0.0
41	EA_Model.Class_Model.atm.physical.Cancelled	N	N	0.0
42	EA_Model.Class_Model.atm.transaction.CardRetained	N	N	0.0
43	EA_Model.Class_Model.banking.AccountInformation	N	N	0.0
44	EA_Model.EARootClass	N	N	0.0

Generate Diagram : Package Diagram | Class Diagram

Figure 9.3: *The SAAbs Tool Displaying Ranking of Classes*

9.3.4 Process: Classification

As a next step, we build and validate scoring models, and apply these to the classes to generate a rank score for importance. We have experimented extensively with a range of machine learning algorithms in prior work, which resulted in the choice of the Random Forests algorithm as the most reliable and robust algorithm for this task ([127], [128]). For the training set, we use 50% of the data for the training and the other 50% for testing. Since the score can only be produced for classes in the test set, we switch the training set to the test set, and the test set to the training set to get a complete score for every class in the class diagram.

9.3.5 Output: Class Ranking

There are several cases where multiple instances share the same score. Therefore, we improve this ranking by using the coupling and number of coupling measures. We rank these instances by prioritizing the instances that have highest total number of coupling measures. The ranking is followed by the number of operations if the total number of coupling measures is the same. An example of the prediction score and the rank of classes is shown in Figure 9.3.

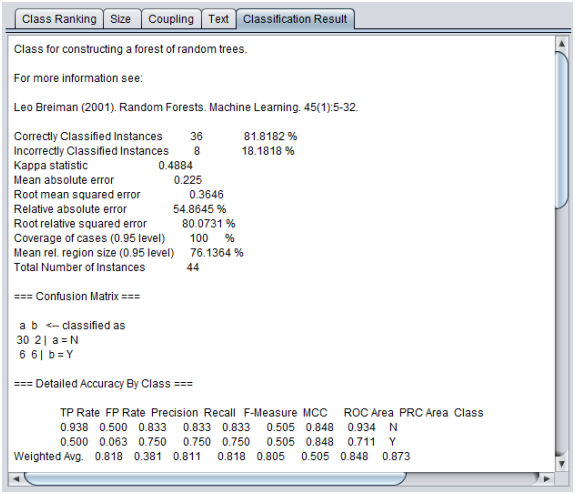


Figure 9.4: *Textual Results of Classification*

9.3.6 Output: Visualization

The visualization in the SAAbs tool is divided into two sections: (i) Prediction Information, and (ii) Architecture Viewer. The details are the following.

Prediction Information

The prediction information displays information about all features used for prediction and the classification results (as demonstrated in Figure 9.4). The values of all features (UML design metrics and text metrics) are displayed to provide the information about the system-to-analyze. The tool also provides the result of the class ranking and also the evaluation measures of the classification algorithm. The Area Under the ROC Curve (AUC) are displayed to show the prediction performance of the classification algorithm.

Architecture Viewer

The SAAbs tool offers users to view a system-to-analyze according to the amount of important classes the user desires to display. It also offers to zoom in/out the class diagram of the system-to analyze. Also, it can limit the information to different levels of detail by hiding classes elements. The details of the Architecture Viewer visualization of SAAbs tool are the following:

1. *SAAbs Slider: Displaying System-to-analyze Architectural Abstraction*
   By introducing a slider, the SAAbs tool offers the architecture to be viewed in the form of class diagrams or/and package diagrams. The generated class diagram

displays all the information (classes, operations, attributes and relationships) while the package diagram shows a combination of the class diagram and the package diagram (without operation and attribute; the relationship of packages is presented based on the relationship between classes). In detail, the following views are offered:

(a) *Hide Classes*: Display important classes based on the selected level of detail (as a percentage). Classes that are less important are hidden. With this feature, class and package diagrams can be simplified. Examples of the SAAbs viewer Graphical User Interface (GUI) are illustrated in Figure 9.5.

(b) *Color Classes*: Display all classes in class/package diagrams. The tool differentiates the important and less important classes in the class diagram by highlighting the less important classes in red (example in Figure 9.6). Another option (*Grayscale coloring*), the tool highlight the less important classes in gray. In this option, the less important classes gradually turn darker to indicate that the classes are less important than others (example in Figure 9.7). The SAAbs tool also has the option to color all classes (in gray) to illustrate the class importance.

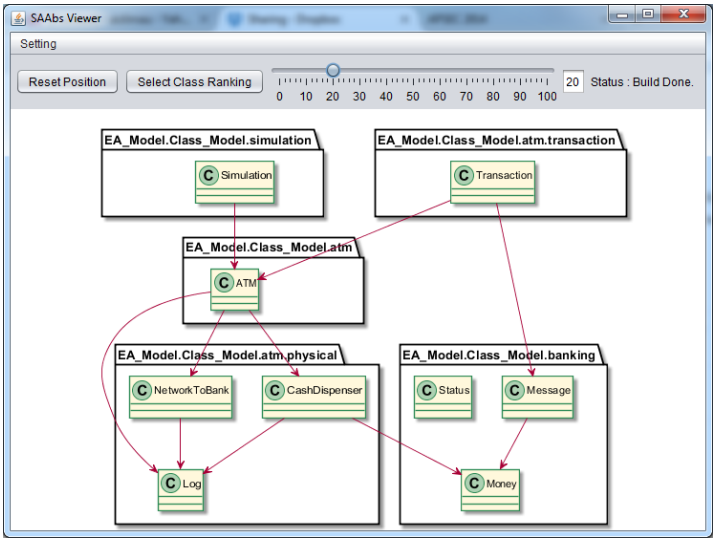
2. *Hiding Class Diagram Elements*: In order to simplify the class diagram, the respondents in [133] have mentioned that some class diagram elements may be hidden: a) getters and setters, and b) constructor. Thus, SAAbs offers the option to hide the getters and setters, constructors, and also attributes and operations.

3. *Viewing Diagrams*: The SAAbs tool provides basic diagram viewing features which are: a) Zooming in and out, b) Zoom in a selected area and c) Class Diagram Rotation. These features allow the users to focus on specific parts of the viewed diagram.

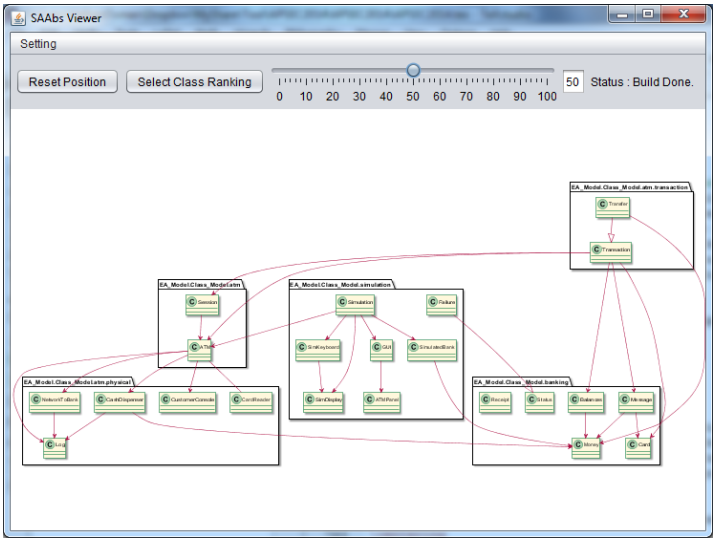
9.3.7 Implementation

The SAAbs tool has been developed using the Java programming language. Netbeans 7.3 [115] was used as the integrated development environment (IDE) for building this tool. We used SDMetrics (Version 2.3-Academic license) for object-oriented metrics extraction and the SDMetrics OpenCore (version 2.31) library for parsing the XMI files. To produce the text dictionary, RapidMiner [11] is used for processing the class names (however, this tool is not yet integrated with the SAAbs tool). The Waikato Environment for Knowledge Analysis (WEKA) [76] library is used for the machine learning classification purposes.

To visualize the output, PlantUML [10] is used for generating the class and the package diagram. PlantUML is a component that renders UML diagrams as Scalable Vector Graphics (SVG). SVG offers an excellent quality for zoom-in and zoom-out. For more details, the tool is available at [134] and the demonstration can be found at [124].



(a) 20% of Abstraction



(b) 50% of Abstraction

Figure 9.5: *SAAbs tool Viewing Class+Package Diagram in Different Level of Abstraction*

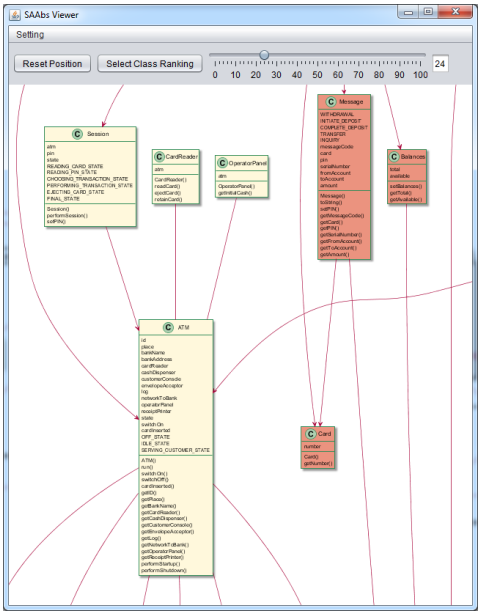


Figure 9.6: *Highlighting of Less Important Classes*

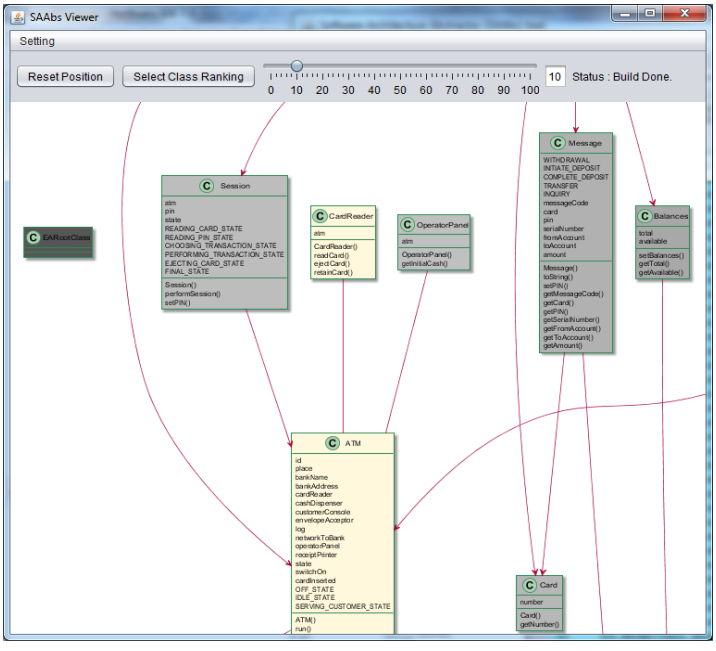


Figure 9.7: *Greyscale Coloring: Less Important Classes with Darker Shades of Gray.*

9.4 Discussion

In this section, we discuss the SAAbs as a Software Exploration Tool. In general, there are three main activities in the reverse engineering process: Data gathering, Knowledge organization and Information Exploration. According to Tilley et al. [167], information exploration holds the key to program understanding. Storey et al. [157], outlined the cognitive designed elements for software comprehension (as illustrated in Figure 2.2). As mentioned in the introduction of this chapter, we would like to provide a tool that fulfilled the following cognitive elements.

9.4.1 **E3**: Provide abstraction mechanism

With the multilevel abstraction feature within the SAAbs tool, software developers abstract the class diagram to any preferred level, by using a "Slider" that can be used to effectively control the level of abstraction, or coloring of the classes. In this way, the software developer can learn the software design from the bottom (a high number of classes) to up (a smaller number of (important) classes).

9.4.2 **E4**: Support goal-directed, hypothesis-driven comprehension

Goal-directed and hypothesis-driven approaches to understanding are normally used in top-down comprehension. Top-down comprehension is suitable for users that are familiar with the system domain. The SAAbs tool facilitates this requirement by allowing users to manually select the important class candidates that they want to focus on.

9.4.3 **E5**: Provide overviews of the system architecture at various levels of abstraction

Viewing a system at various levels of abstraction is needed for top-down software exploration. The "Slider" in our tool can generate multiple class diagram at various levels of abstraction. It can be used for a bottom-up approach as well as for the top-down approach. Furthermore, our tool also can generate the abstraction of software designs in different (separated) windows for different focus of abstraction (i.e. different "ground truth") for comparing abstraction results.

9.4.4 **E6**: Support the construction of multiple mental models & **E11**: Show the path that led to the current focus

Often, different users require different views on the same system. Therefore, a facility to construct multiple models is needed to cater for the needs of different users. Because UML diagrams are our main focus, the SAAbs tool provides two types of diagrams:

Class Diagrams and Package Diagrams with multiple levels of detail. Users can exclude the attributes, operations, getters/setters and constructor methods. This provides users with some options to construct a class-based views based on their own preferences. The SAAbs tool also provides a facility to show all classes of the system through colouring or transparency highlighting subsets of classes that are identified as important.

9.4.5 **E15**: Provide effective presentation style

Our tool provides multiple options for software design representation that purposely to reduce the software design complexity. The SAAbs tool provides two main features to reduce the complexity of software design: 1) The condensation of class diagrams (multilevel levels of abstraction) can reduce the complexity of a design by leaving out less important class diagram and reduce the amount of relationship and, 2) Highlighting the important classes to emphasize the classes that the users should focus on.

9.5 Conclusion and Future Work

This study presents a tool-supported framework for visualizing a software system at various levels of abstraction. The SAAbs tool includes features to automatically predict the importance of classes in a class diagram. It also provides a scalable and interactive visualization of software systems. Abstraction of class diagrams, highlighting classes and multiple levels of details are the main features of the tool. This research discusses how this tool links to several cognitive design elements for program comprehension.

We validate the effectiveness of our proposed framework by studying the usefulness of the tool in assisting software developers in a program comprehension task. The validation results are presented in Chapter 10.

The presented tool is an early development of implementing our findings in this research. We consider several ways to enhance and improve the SAAbs tool. Amongst possible future work are the following:

1. Improve the layout option to be more interactive and dynamic; provide users options to modify the layout of the diagram dynamically and provide several automated layout options (hierarchy, centralization).

2. Provide interactive learning or semi-structured learning options. In this way, the abstraction of class diagrams can be done interactively and real-time based on the feedback from the users on which classes should be included or excluded (i.e. by changing the “ground truth” in real-time).

Part III

Validation and Conclusion

Chapter 10

Validation

This chapter describes a study to validate the Software Architecture Abstraction (SAAbs) framework. This framework is purposely invented to simplify reverse engineered class diagrams by selecting the important classes in a system. Using the condensed class diagrams generated by the SAAbs tool, we validate the SAAbs framework based on the user view of these class diagrams. This study focuses on the usefulness of the condensed class diagrams for program comprehension.

10.1 Introduction

In this chapter, we present the validation of the SAAbs framework to enhance software comprehension of the RE-CD. Based on the previous chapters, we have found that Random Forests is the most suitable classification algorithm for this approach. Based on this, we have developed a tool (called Software Architecture Abstractor (SAAbs) tool - described in Chapter 9) to apply our approach to the RE-CD. As a result, the tool produces a ranking of importance classes for all classes in a class diagram. A higher score indicates that the class is important. With this ranking, condensations of a class diagram based on the importance of classes can be constructed. By using the condensations of the class diagrams, we carried out a survey to validate our approach. This validation aims at i) discovering the understandability of condensed class diagrams, ii) finding whether the condensed class diagram generated by this approach is helpful in understanding the software design and, iii) validating the SAAbs tool in assisting software developers to understand the software.

This chapter is part of a publication entitled **"Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams"**, In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014)

The chapter is structured as follows. Section 10.2 describes the research questions and is followed by Section 10.3 that explains the experiment design. Section 10.4 demonstrates the results and we discuss our findings in Section 10.5. This is followed by the conclusions and our suggestions for future work in Section 10.6.

10.2 Research Question

In this chapter, we aim at answering the following question:

> **MQ:** *Can the SAAbs framework help the software developers for system comprehension?*

In order to answer the main question, four research questions need to be explored. The research questions are the following:

- **RQ1:** *What is the level of understandability of the condensed class diagrams created by SAAbs framework?*
- **RQ2:** *What level of abstraction is preferred to produce an overview of a software design (in particular, based on the respondents' background)?*
- **RQ3:** *Does the condensed class diagram represent the important information out of the reverse engineered class diagram?*
- **RQ4:** *What is the relative usefulness of SAAbs tool for different respondents' background?*

10.3 Experiment Design

In this section, we describe the study design. We explain the design of the questionnaire and how we conduct the study.

10.3.1 Questionnaire Design

The questionnaire is divided into four parts, which are: Personal Background, Choices of Class Diagram, Software Architecture Abstraction (SAAbs) framework and The SAAbs tool. The explanations of these parts are the following.

Part A: Personal Background

In this part, we collect information about the respondent background. The respondents were asked about their skill, experience, frequency of using UML class diagrams and also their experience in reverse engineering source code into UML Class Diagrams. This information is used to discover the relation between the respondents' background and their answers.

Table 10.1: *Type of Class Diagram used in this study*

Class Diagram	Type of Class Diagram	Description
CD.FD	Forward Design	Displays the forward design
CD.25	25% condensation of the RE-CD	Displays only 25% of classes in the class diagram and excludes the other classes.
CD.50	50% condensation of the RE-CD	Displays only 50% of classes in the class diagram and excludes the other classes.
CD.75	75% condensation of the RE-CD	Displays only 75% classes of classes in the class diagram and excludes the other classes.
CD.RE	RE-CD	Shows 100% of all reverse engineered classes in the class diagram

Part B: Choices of Class Diagram

This part aims at discovering the level of understanding of class diagrams generated by the SAAbs framework. We used an ATM simulation system [28] as a sample case. We describe the ATM system requirement at the beginning of this question. Then, we provide five class diagrams in which the respondents are obliged to rate all presented class diagrams in term of design comprehension. Detailed information about the class diagrams is shown in Table 10.1.

The ATM simulation project in [28] provides a complete UML diagrams. We use the forward design of this system (CD.FD) and we reverse engineer the ATM simulation system source code into a class diagram to produce the CD.RE. In order to produce the condensation of the RE-CDs (CD.25, CD.50 and CD.75), the following information is used:

- the RE-CD (in XMI) as the diagram to be condensed.
- the analysis phase class diagram as the input for the important classes candidate.

We load all information into the SAAbs tool to generate the ranking of the important classes in the class diagram and also reconstruct CD.25, CD.50 and CD.75.

In this study, the respondents are required to give opinions on their understandability (based on a 6-point scale) of all provided class diagrams (see Table 10.1). They also need to choose the preferred class diagram for program comprehension and give their suggestion(s) to improve the presented class diagrams. This part is formulated to answer RQ1 and RQ2.

Part C: Software Architecture Abstraction (SAAbs) Framework

This part aims at verifying the condensation result from the SAAbs framework. We want to investigate whether the condensed class diagram generated by the SAAbs framework represents the important information out of the RE-CD; and investigate whether this information (important classes) is useful to understand the system.

A Pacman game [44] was used as a case study of this part. For this question, we provide: a) the explanation of the game at the beginning of this question, b) A Pacman game RE-CD, and c) A 50% condensation of the Pacman game RE-CD. The Pacman's game RE-CD was constructed by using the latest version of the source code (version 4). Then, a 50% condensation of the RE-CD was created by using the SAAbs tool. The 50% of condensation is chosen to represent the condensation of the RE-CD because the number of classes that appear is not too small and not too large. We condense this diagram by using the first version (release) of forward design of this project as candidates of important classes.

The respondents are asked to study the class diagrams and give their judgment in terms of the usefulness of condensed RE-CDs in understanding the system. This part is formulated to answer RQ3.

Part D: The SAAbs Tool

This part aims at validating the SAAbs tool. We asked the respondents to give their opinion on the tool in assisting them for software comprehension. Respondents were also asked to indicate the best feature of the tool and suggestions for improvement and enhancement of the tool. This part will answer RQ4.

10.3.2 Experiment Description

In this subsection, we clarify how the experiment is conducted. The flow of this experiment is shown in Figure 10.1. The respondents were selected from different types of background, i.e. students, academic researchers and IT professionals. Depending on the respondent's background, we asked the respondents to answer different questions. For students, they are obliged to answer Part A, B and C, and graduate students (academic researcher, IT professional) need to answer all questions. The respondents are divided into these groups because we are focused on the experienced respondents in the software implementation in evaluating the tool in Part D.

The answers were written on the prepared response sheets. The respondents may answer the questions in Part A, B and C directly. However, before answering Part D, the respondents are given a "live" tool demonstration (individual or in a small group). The respondents are offered to use the tool or to load their own input to the tool after the presentation. We also conducted an informal question and answer session to make sure that the respondents have a detailed understanding of the tool. Then, the respondents are allowed to answer Part D.

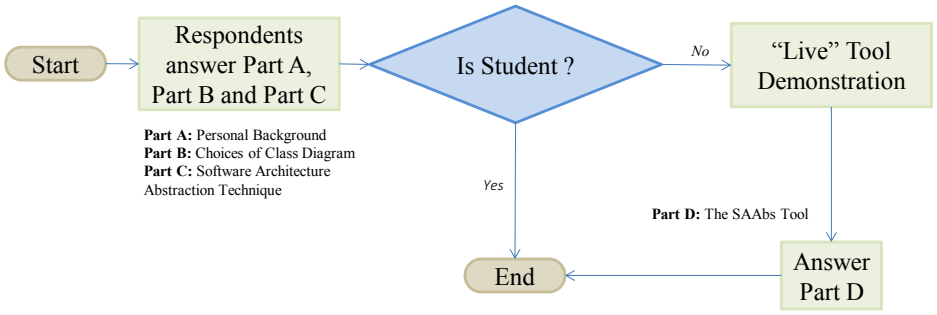
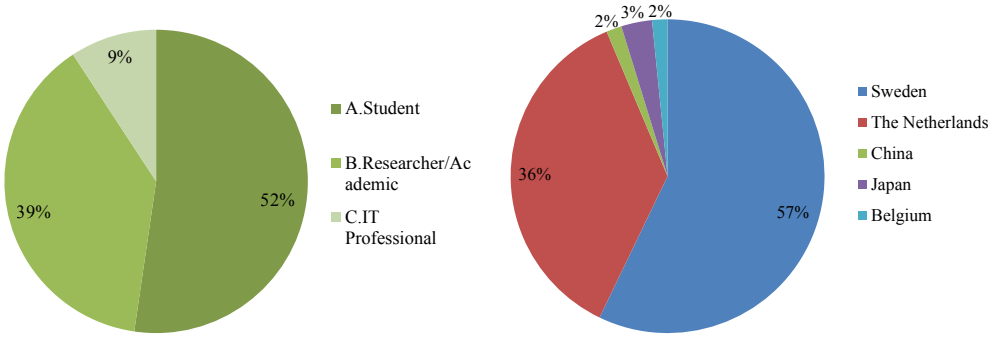


Figure 10.1: *Flow of the Experiment*



(a) Respondents' Role (b) Respondents' Location

Figure 10.2: *Distribution of the Respondents*

10.4 Results

In total, we received 65 responses to this survey. However, due to incomplete of answers, we only count 63 respondents. The answered questionnaires can be found at [121]. The respondents are coming from three different roles which are i) Students (52%), ii) Academic Researcher (39%) and iii) IT professional (9%). The majority of the respondents are from Sweden and the Netherlands, and other locations are China, Japan and Belgium. The distribution of the respondents' role is illustrated in Figure 10.2. In terms of the respondents' experience in class diagrams, 27% of the respondents have < 1 year of experience while another 36% of the respondents have in between 1 to 3 years (see Figure 10.3). 35% of the respondents have more than 3 years of experience in class diagrams. In Figure 10.4, we show the skill of the respondents in understanding the class diagram. This figure demonstrates that 76% of the respondents have an *average* and above skill in understanding class diagrams while in total 22% of the respondents have *low* or *poor* skill in understanding class diagrams.

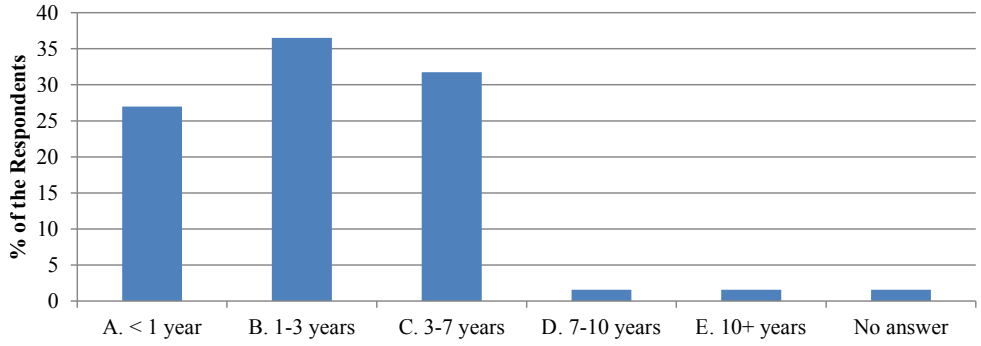


Figure 10.3: *Respondents' Experience with Class Diagram*

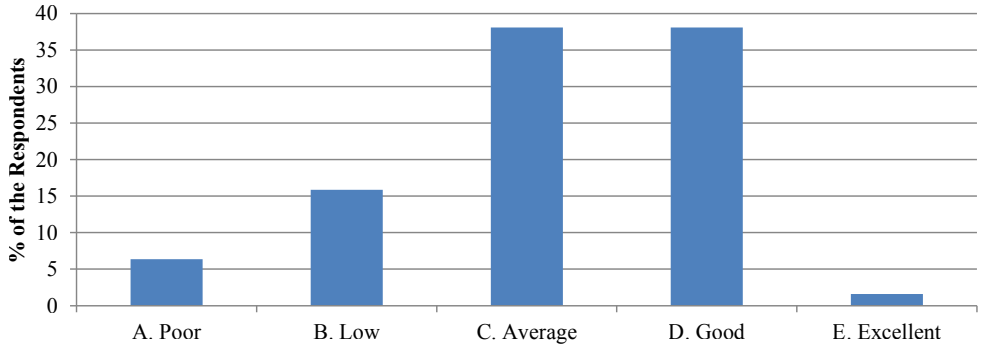


Figure 10.4: *Skills of Understanding Class Diagram*

In terms of the usage of class diagrams, 14% of the respondents *do not use* class diagrams at all for software comprehension. 33% of the respondents use class diagrams *if required*, and 43% of the respondents sometimes use class diagrams. Only 10% of the respondents frequently use class diagrams for software comprehension.

In terms of RE-CD, the majority of the respondents (70%) has *low* or *poor* experience in RE-CDs, while 30% indicated that they have an *average* and *good* experience in reverse engineering source code into class diagrams. The remaining results are presented based on the research questions mentioned in Section 10.2.

10.4.1 RQ1: The Understandability of Condensed Class Diagrams

Figure 10.5 shows the overall results of the understandability of condensed class diagram. The detail information on CD.FD, CD.25, CD.50, CD.75 and CD.RE is explained in Table 10.1. We found that one of the condensed RE-CDs generated using our approach is more understandable than the FD. The CD.25 rated on average 4.58 points (out of 6) compared to CD.FD that is rated on average 4.08 points. CD.RE has been

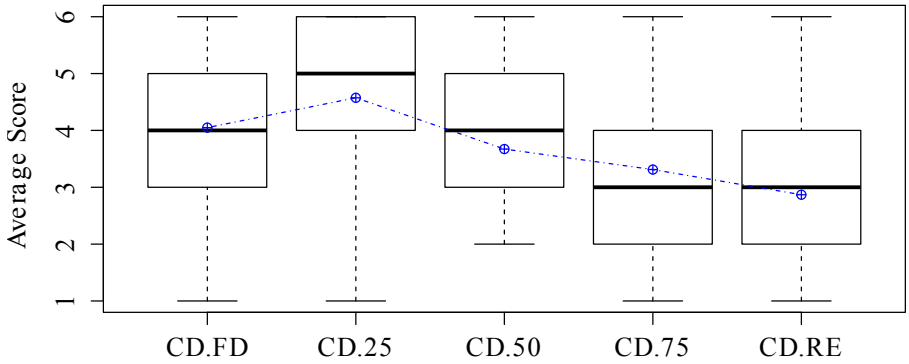


Figure 10.5: *The Understandability of Condensed Class Diagram*

rated 2.87 points on average that makes this diagram has the lowest rating from the respondents. This result shows that condensed RE-CDs scored better ratings than the RE-CD. On the other hand, the line chart (in Figure 10.5) indicates the understandability rating decreases when the number of classes increases. Therefore, we believe that the number of classes influences the respondents' judgment. In the next subsections, we further investigate the rating results by relating the respondents' background and their average rating.

Respondents' Role vs. Understandability Score

Figure 10.6 illustrates the average rating for each class diagram based on the respondents' role. The overall result shows that the rating of CD.25 is higher than the CD.FD. However, referring to Figure 10.6, the IT professional prefer the forward design (CD.FD) over the CD.25. It is quite clear that the rating of the class diagram is decreasing according to the number of classes for the academic researcher group. However, the average rating for students for CD.50 and CD.75 is almost equal.

Respondents' Experience vs. Understandability Score

From the perspective of the respondents' experience, the results show that respondents that have 3 - 7 years experience in class diagrams prefer forward design (CD.FD) compared to condensed class diagrams (CD.25, CD.50, CD.75) and the CD.RE. The results (see Figure 10.7) also show that the average rating for CD.50 and CD.75 is almost equal for the respondents that have > 1 year experience in class diagrams.

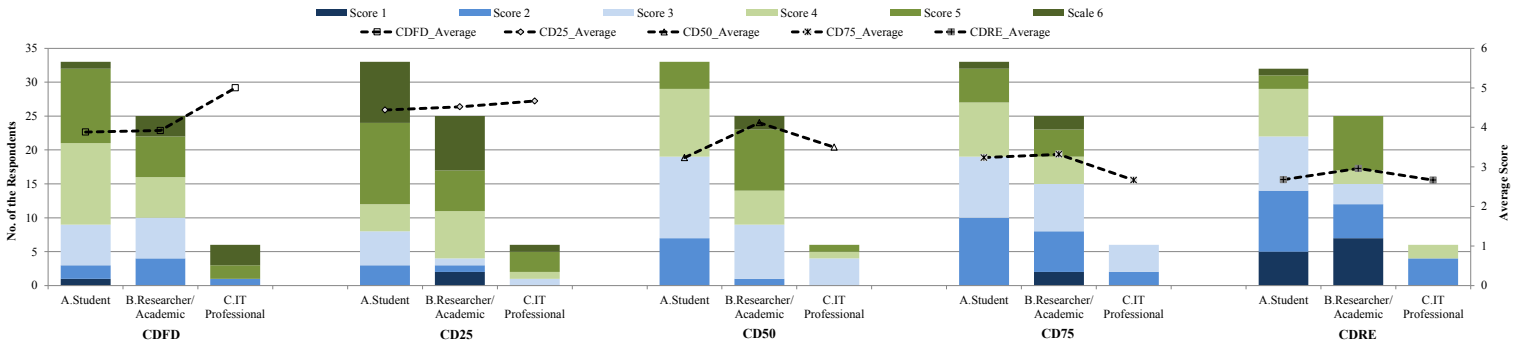


Figure 10.6: *The Respondents' Role vs. Understandability Score*

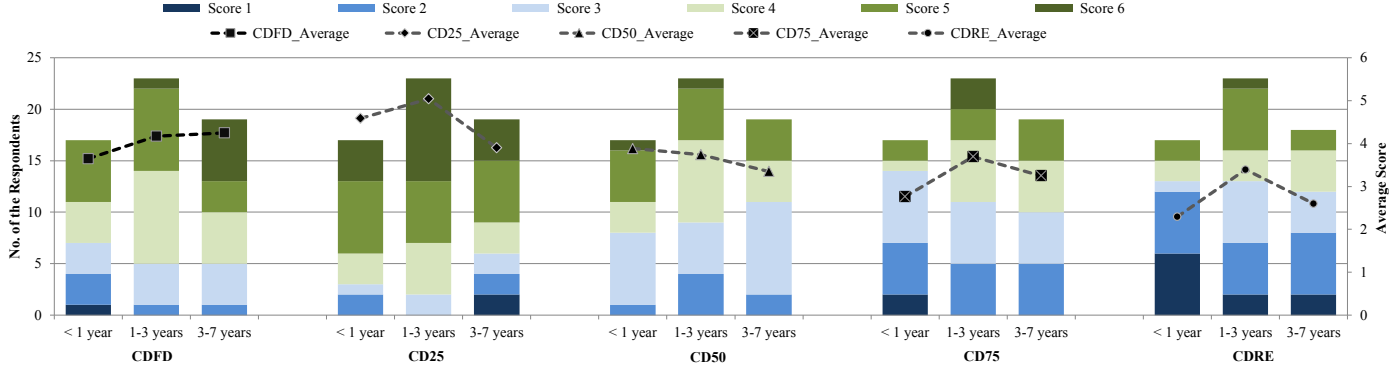


Figure 10.7: *The Respondents' Experience vs. Understandability Score*

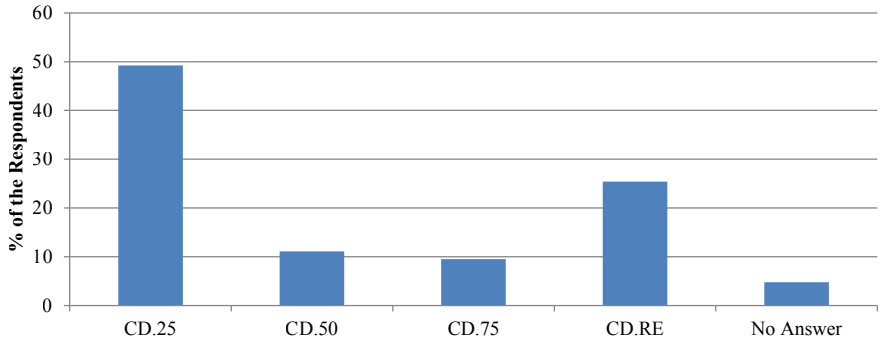


Figure 10.8: *Choices of Class Diagrams*

10.4.2 RQ2: Choices of Class Diagram

In the previous question, we assess the understandability of the condensed class diagrams. In contrast, this question is intended to compare the respondents' preference between the condensed RE-CDs (CD.25, CD.50, CD.75) and the RE-CD (CD.RE). We want to know which class diagram is preferred to be used for understanding the system. The forward design is not offered as an option because this diagram is most likely will be chosen by the respondents as the preferred class diagram. The respondents were asked to choose the class diagram that they preferred for software comprehension. Therefore, the respondents may only choose those between 4 diagrams and they also need to explain their motivation behind their choice. The overall results of the respondent's choices are demonstrated in Figure 10.8.

Based on the responses in RQ1, we expected that the number of classes presented in the question will play a significant role in choosing the class diagram. However, it is interesting to see that the number of classes does not seem to influence the decision in choosing a class diagram. The results demonstrated that CD.25 (the smallest amount of classes) and CD.RE (the highest amount of classes) are two main class diagrams chosen by the respondents. Figure 10.8 shows that almost half (49%) of the respondents chose CD.25 as their preferred class diagram for viewing the system. In total, 31 of the respondents prefer the class diagram in CD.25 for system understanding. 42% of these respondents prefer this class diagram because it is small, simple and easy to look into. 32% of these respondents mentioned that this class diagram showed an appropriate level of condensation of the class diagram that show important classes in the class diagram.

For the CD.RE, 25% of the respondents prefer this diagram for system understanding. Completeness is the main reason for choosing this class diagram. 63% of the respondents that prefer this diagram mentioned that this class diagram is complete and shows detailed information. 19% of these respondents stated that this class diagram is compact, concise and comprehensive. Even though only 6 respondents (11%) chose the

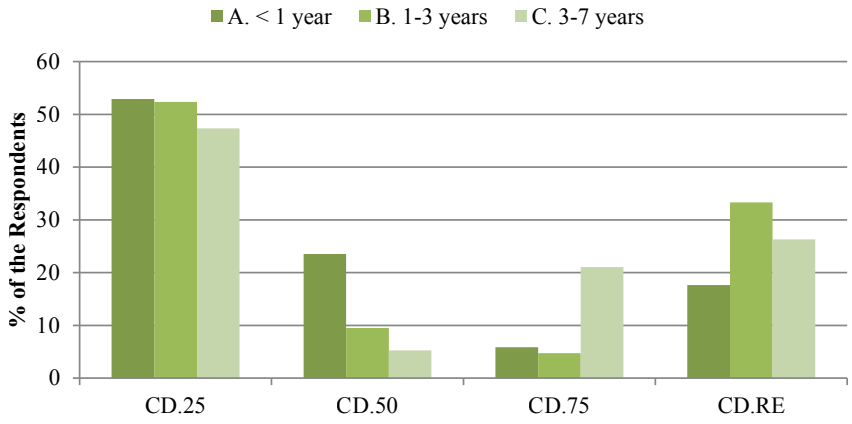


Figure 10.9: *The Respondents' Experience vs. Choice of Class Diagram*

CD.50, 85% of these respondents indicate that this class diagram has the appropriate level of condensation that helps the overview of the system.

Respondents' Experience vs. Choices of Class Diagram

From the perspective of respondents' experience, the results in Figure 10.9 show that from 25% of condensation to 50% of condensation the percentage of the respondents that chosen these diagrams is decreasing and start increasing from CD.75 to CD.RE. Figure 10.9 also suggests that the more experienced respondents chose class diagram with a higher amount of classes.

10.4.3 RQ3: Software Architecture Abstractor Framework

In part C, we asked the respondents to compare the RE-CD and the condensed RE-CD to know how useful this diagram (generated by the SAAbs tool) in representing the important information in the RE-CD. Out of 63 respondents, 5 respondents did not answer this question. Figure 10.10 shows the results of this part. On average, the respondents have given a rate of 4.72 on a scale of 1 to 6. This indicates the 50% condensation of RE-CD constructed by using this framework is useful for software comprehension. 67% of the respondents rated 5 and above while 9% of the respondents rated below 4.

10.4.4 RQ4: Usefulness of the SAAbs Tool

In total, 29 respondents answered this question (only academic researcher and IT professional). As can be seen in Figure 10.11, 93% of the respondents have given the

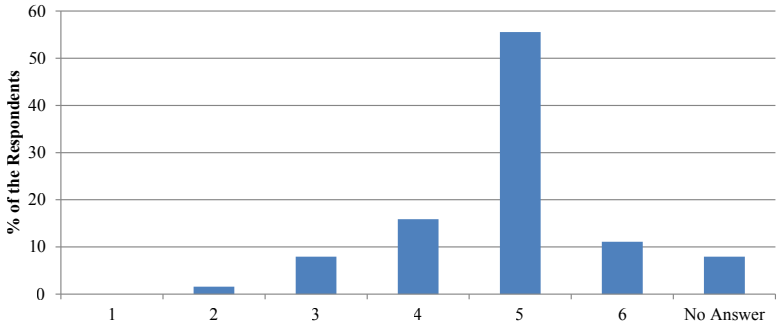


Figure 10.10: *Level of Abstraction of RE-CD for Software Comprehension*

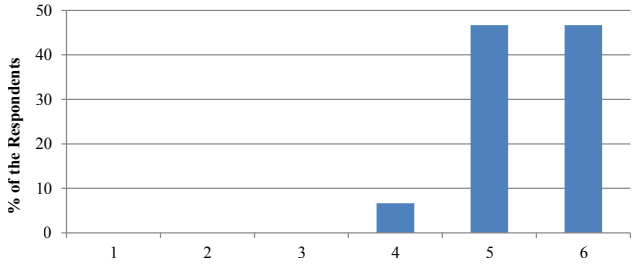


Figure 10.11: *The Rating of the Usefulness of SAAbs Tool*

rate of 5 and above while 7% respondents have given the rate below 5. On average, the respondents give 5.40 out of 6 points. This suggests that the SAAbs tool may be useful for understanding a system.

The respondents were also asked to answer the reason for the score given to the tool. As a result, 50% of the respondents mentioned that the tool is able to show the multiple levels of condensation. 40% of the respondents mentioned the ability of the tool to show the important classes (e.g. highlighting with color) and 40% of the respondents mentioned that this tool is helpful for showing an overview of systems. The remainder of this section describes the results in detail.

SAAbs Features vs Respondents' Background

We further analyze the relationship between the choices of the best features and the respondent's background information. The respondent's background covers the respondent's role, skill and experience in using class diagrams. The respondents' experience only includes group < 1 year, 1-3 years and 3-7 years of experience because the majority of the respondents (97%) is coming from those groups. The reasons mentioned by the respondents are analyzed by capturing the keywords of their answers. Then, we group these keywords to the related features. Overall, four main features which are mostly mentioned by the respondents: (1) Multilevel Class Diagram Abstraction, (2)

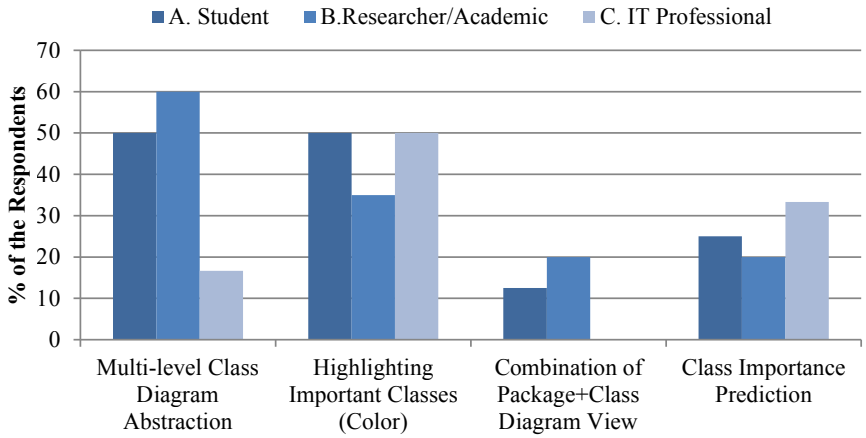


Figure 10.12: *The Respondent's Role vs the Tool's Features*

Highlighting Important Classes, (3) Combination of Package and Class Diagram View and (4) Class Importance Prediction. Hence, we only analyze those four mostly chosen features.

Role vs. Features
As can be seen in Figure 10.12, the academic researcher group prefers the Multi-level Class Diagram condensation over Highlighting (Coloring) important classes. In contrast, the IT Professionals like the Highlighting of Important Classes more than the Multi-level Class Diagram Abstraction.

Experience vs. Features
Figure 10.13 shows the relationship between the respondent's role and their choices of most valued features. It is interesting to see that the preference of Multi-level Class Diagram Abstraction decreases when the years of experience in using the class diagrams is increased. This may indicate that the more experienced respondents are, the less likely they want to simplify the diagram. On the other hand, the preference for Highlighting Important classes increases with the years of experience in using class diagrams. This result suggests that highlighting important classes is preferred when a software developer becomes more experienced. However, both features are of equal interest to (60%) respondents that have 1-3 years of experience. Figure 10.13 demonstrates that the respondents are slightly more interested in the Combination of Package+Class Diagram view as the years of experience are higher.

Skill vs. Features
Figure 10.14 provides the overall result of the relationship between skill in understanding class diagram and the tool's features. There is not much different for the

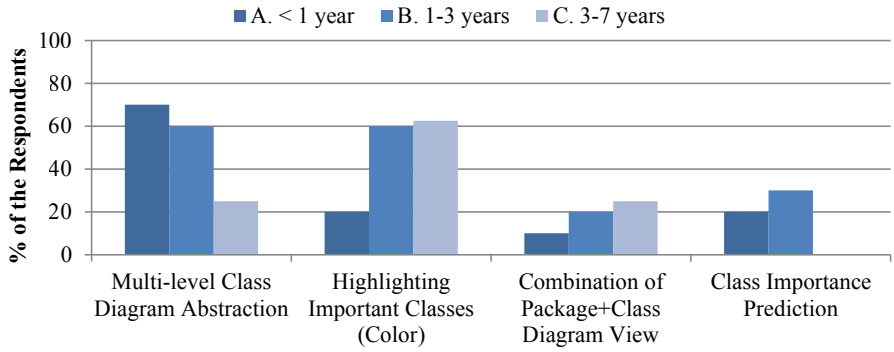


Figure 10.13: *The Tool's Features vs the Respondent's Experience*

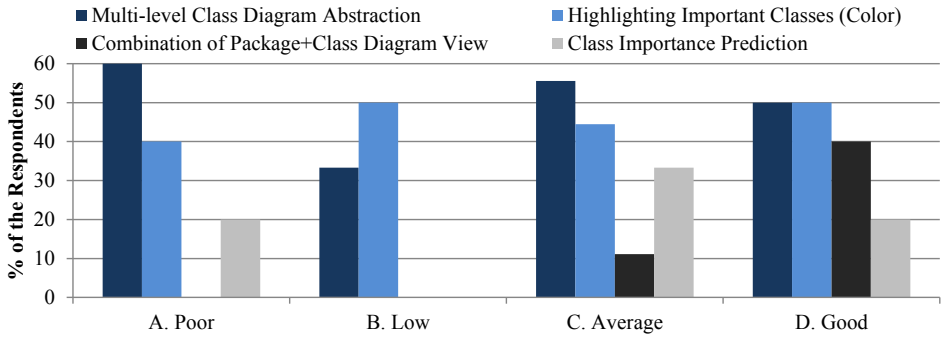


Figure 10.14: *The Respondent's Skill vs the Tool's Features*

Multi-level Class Diagram Abstraction and Highlighting Class Diagram features for all levels of skill. However, there is a small percentage that shows that the Combination of Package+Class Diagram View is more attractive for the respondents that have the skill of *average* and *good* in understanding class diagrams.

Limitations and Improvements

To improve this tool, we asked the respondents about: (1) the weakness of the tool and (2) the improvement needed for this tool. In general, both results shows that the enhancements or improvements needed for this tool mainly referred to: (1) Important Classes Prediction, (2) Layout and (3) Additional Tool's Features. Figure 10.15 shows the results on the limitations of the tool.

For Important class prediction, the respondents show their concern about the prediction result. They are concerned about the suitability of the machine learning techniques in classifying the important classes of "big systems" and also the validation of the result (human validation of the important classes prediction of the tool). In terms of layout, the respondents suggested that the tool should give more options to the

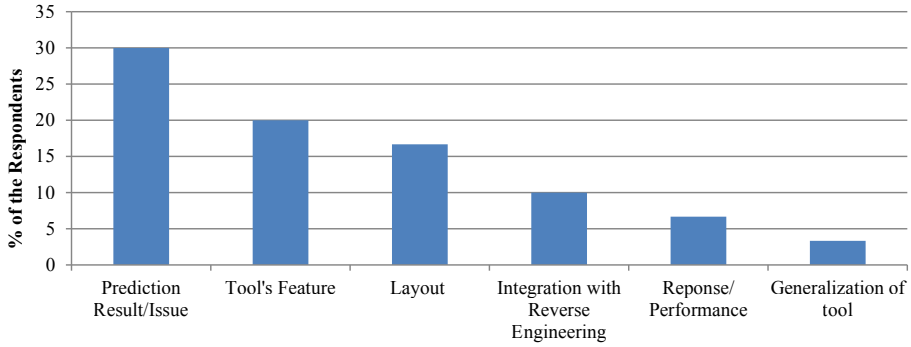


Figure 10.15: *The Limitations of the SAAbs Framework and Tool*

tool's user to modify the layout. The modification includes: editing class diagrams, change of color, various types of class diagram layout (e.g. centering, hierarchy) and highlighting of private and public attributes.

10.5 Discussion

In this section, we discuss the scoring of class diagrams, the limitation of SAAbs tool and the threats to validity.

10.5.1 Choosing a Class Diagram

At the beginning of our analysis, we expected that the respondents preferred class diagrams that have a low number of classes. However, our expectation applied for CD.25 but the other preferred class diagram is CD.RE. This result shows that the number of classes is not the main reason for choosing the class diagram. Also, the representation of the class diagram also influences the decision of choosing the best class diagram for system understanding.

According to the reasons of choosing a class diagram, the respondents indicate that the chosen class diagram is easy to understand. Even though CD.RE shows the highest number of classes, the respondents mentioned that this class diagram is understandable. One of the reasons is that the class diagram presents the separation of features where the two major classes are presented in the middle of the class diagram. Those classes represent two independent groups of functionalities that are ATM related system and the simulation/graphical user interface (GUI) related purposes (see questionnaire in [121] for more detail). This suggests that class diagrams with a high number of classes are practical for software comprehension provided that the layout shows suitable clustering or grouping of functionality (as suggested by the software developers in Chapter 5).

10.5.2 Limitation of SAAbs

As described in the previous sections, we proposed the SAAbs framework and tool to assist the software developer for the software comprehension assignment. However, several limitations have been identified before and after the validation experiment was conducted.

For important class prediction, object-oriented design and text metrics are used as predictors. As mentioned in [127], combining those metrics results in a better prediction than using only one particular set of metrics for important class prediction. However, the text metrics are limited to systems that are developed using the English language (for the class element name); Thus, if a system is developed based on another language, these predictors need to be adjusted. This problem due to the stemming algorithm that is used to formulate the text metrics.

Although the object-oriented design metrics have reasonable predictive power, we are considering other features to enhance the prediction results. For instance, the use of network metrics as described by Thung et al. [164].

Another concern about the framework component is the usage of "Show Suggestion" to suggest the candidates for important classes. This suggestion may not be correct because it is based on the amount of coupling and number of operations. It may include several lookup/descriptor classes (like a lookup table) that have a lot of relationships, but these classes are not frequently considered as important classes. In the SAAbs tool, the selected important class candidates are shown in the class list. Users can select the lookup/descriptor classes and other insignificant classes to be excluded from the important class candidates.

10.5.3 Threats to Validity

In this subsection, we discuss the internal and external threats to validity of the validation experiments.

Internal Validity: The selection of case studies may influence the result of this experiment. The respondents need to comprehend a system in a limited time. We believe that we have minimized this threat as we used a small to medium size of class diagrams, well-known projects domain, and the respondents were given a briefing and comprehensive description of the projects.

In this experiment, we tried to discover the suitable features of the tool in regard to the respondents' background. We asked the respondents to indicate the best feature of the tool. We realized that the term "best feature" in the question might not precisely indicates suitable feature. Nonetheless, we believe that the question yields the interest of the respondent on features that are considered suitable features for a particular respondent.

External Validity: The distribution of the respondent's role could be a threat to external validity. The role of respondents is not equally distributed where the majority of the respondents are students. However, we have shown that the distribution of the students and graduate students (academic researcher and IT professional) is almost equal. Although we do not deny that the experience in industry and academia are different, in terms of class diagram understanding, we believe that the difference is small.

10.6 Conclusion and Future Work

This experiment aims at validating the SAAbs framework and tool in assisting the software developer in understanding software systems. We showed that the condensed RE-CDs produced by the SAAbs tool (particularly 25%, 50% and 75% of condensation) are understandable. Furthermore, the respondents considered that the tool that realized the SAAbs framework is useful in aiding the software developer to understand software.

From the results, it is interesting to see that the main class diagram (that is chosen for system overview) is the 25% condensation of RE-CD and the RE-CD (full RE-CD). This result shows that the layout influences the selection of a class diagram for viewing the system rather than the number of classes in class diagrams. The respondents indicate that the condensed RE-CD is also useful for understanding the system. On average, the respondents give 4.72 out of 6 points that indicate the condensed RE-CD is useful for software comprehension.

The SAAbs tool was developed to realize the SAAbs framework in providing a platform for assisting software comprehension. In this experiment, we assess the respondent's judgment on the usefulness of this tool. On average, the respondents gave 5.4 out of 6 points that indicate this tool is helpful during the software comprehension. The result also demonstrated that the respondents that have more experience intend to view all classes in the class diagram and highlight the important classes. In contrast, the less experienced respondents prefer to limit the classes in the class diagram by reducing the class diagram for software comprehension task.

This validation experiment shows an interesting and promising result. There are several ways to enhance and strengthen this validation. We are looking forward to improving the validation by:

- A user study on the usage of the SAAbs tool in the real software comprehension task (task-oriented validation), and
- Online user validation (i.e. publish the tool online and assess the user experience about the tool).

Chapter 11

Conclusions

UML is recognized as the standard for describing software designs. Keeping UML designs up-to-date with evolving source code is challenging and time-consuming. For this purpose, automatic recovery of design diagrams in UML notation out of implementation artifacts (i.e. source code, execution files/library) is an attractive option to obtain and maintain up-to-date design representations of systems. However, understanding reverse engineered UML diagrams is often difficult. Hence, this research aims at providing an automated framework to simplify reverse engineered UML diagrams (specifically class diagram) for assisting software comprehension. In this chapter, we summarize the findings based on the research questions posed in Chapter 1 of this thesis. We describe the contributions of this research and outline future work.

11.1 Summary of Findings

The goal of this research is to devise an automated framework for simplifying UML class diagrams to assist the software comprehension task. The following is the main research question that has been formulated to clarify the scope of this research:

Main RQ: *What method of condensing of reverse engineered class diagrams helps developers to understand the design of software systems?*

We decomposed this main research question into five research questions (described in Chapter 1). At the start of our research, we conducted two studies to investigate the usage of UML diagram in open source software development (OSSD) and the state-of-the-art of reverse engineering source code into class diagrams. In doing so, we identified several OSSD projects that are suitable for our research. We highlight

Table 11.1: *Summary of Background Research Findings.*

Study	Findings
UML usage in OSSD	1. In software development, the focus of modeling shifts from an initial focus on structural aspect in the early phases of development towards fleshing out behavioural aspects of the design in the later stages of development. 2. The frequency of updating UML diagrams is low when compared to the frequency of updating source code. As triggers for the updating of models, we identify: i) major changes to the software, and ii) the joining of a group of new developers into the project.
Reverse engineering of source code into class diagrams	1. Existing CASE tools are not able to correctly recover aggregation and composition relationships from the source code. 2. Existing CASE tools are not able to correctly recover/represent the bidirectional relationship. 3. There is a wide variety in the quality of the class diagrams that are obtained via reverse engineering by different CASE tools. Not all CASE tools are suitable for reverse engineering source code to UML class diagrams.

findings of these studies in Table 11.1. Next, we recapitulate our research questions and their main findings.

11.1.1 **RQ1:** *Which information in class diagrams do developers find important for understanding software designs?*

For this question, we conducted a semi-structured survey to gather data about developers' views of classes that could be left out from a class diagram and classes that should remain for good understandability of the system design. In this survey, 32 professional software developers participated.

Our analysis focused on the characteristics of classes that could be left out. We discovered that class relationships, and the role and responsibility of classes play major roles in determining class inclusion/exclusion.

Specifically for class exclusion, we found that library classes and Graphical User

Interface (GUI) classes (esp. when generated automatically by an Integrated Development Environment (IDE)) could be left out from the class diagram. These categories of classes have a small relation to the application domain. In other words, this study shows that for gaining an understanding of a new software system, developers (at least in their initial exploration) focus on classes that are related to the application domain.

The aforementioned findings give us insights into the information of class diagrams that are important for software developers (for inclusion) and information in class diagrams that can be left out (for exclusion) in order to simplify a class diagram. Our subsequent research is to use this information to devise an automated approach to assist software comprehension.

11.1.2 **RQ2:** *Which object-oriented design metrics do developers find most indicative for class importance?*

The second research question aimed to study the relevance of object-oriented design metrics in deciding on class inclusion and exclusion in class diagrams. We addressed this question by performing an online survey involving 25 participants from different types of background (i.e. students, academic researchers and IT professionals).

This research also discovered that the number of public operations (NPO) is the most important object-oriented design metrics in deciding the class inclusion: classes that have a high NPO are more likely to be recommended for inclusion in the class diagram. The findings of this survey suggest that object-oriented design metrics (esp. from the size and coupling category) are relevant features for deciding on class inclusion and exclusion.

11.1.3 **RQ3:** *How to automatically condense class diagrams using object-oriented design metrics?*

We studied the suitability of the object-oriented design metrics as features for predicting class inclusion and exclusion in class diagrams. We applied a machine learning approach to construct a classifier for class inclusion/exclusion using supervised learning methods. Nine open source software projects have been collected as case studies for this.

Our study focused on the application and domain related classes. Therefore, we filter the classes in these projects by removing external library and runtime classes. These projects were selected because they all contain UML designs that are manually created during the forward design. In these cases, we use the classes that exist in the forward design as the 'ground truth', which is used for the training of the classifier. We also compare the performance of nine classification (i.e. machine learning) algorithms to determine the most suitable algorithms for deciding class inclusion/exclusion. These classification algorithms produce a score for every class. This score enables the ranking of classes according to their likelihood of inclusion. Because our datasets are typically

imbalanced, we use the Area under the ROC Curve (AUC) to evaluate the performance of the classification algorithms.

Our findings demonstrate that Export Coupling Parameter (EC_Par), Dependency In (Dep_In) and Number of Operation (NO) are the most influential features in classifying class inclusion/exclusion. The classification that is based on all features (i.e. the 11 object-oriented design metrics) achieves the best AUC value. This means that all features are considered valuable for the class inclusion/exclusion classification. This research also found that Random Forests and k-Nearest Neighbor algorithms are the most suitable for our prediction purpose. For nine case studies, Random Forests produces an AUC score above 0.64 with an average AUC score of 0.74.

It is not reasonable to expect that this approach could produce a 100% correct prediction of class inclusion/exclusion. One reason for this is the imbalance in the dataset which is the basis for learning. Nevertheless, these datasets present realistic scenarios found in software projects. This research has commenced a novel approach of using machine learning based on the condensation of the RE-CDs.

11.1.4 **RQ4:** *Can the automatic condensation of class diagrams be enhanced by using class names?*

Prior research (RQ1 and RQ2) indicated that the role and responsibility of classes conveyed important information about a design. We tried to capture these notions in our prediction by exploring the use of features based on class names. We came up with text metrics based on class names by using text processing methods.

The experiments in this chapter followed the same structure as the experiment for RQ3 where we evaluate each feature's predictive power as well as the feature's performance. Nine classification algorithms (as in RQ3) were used in this experiment and ten OSSD projects (nine of them from RQ3) were used as case studies. We use the result of RQ3 as a reference benchmark to evaluate the improvement of the text metrics in class inclusion/exclusion prediction. We study the effects of using different categories of features, namely: text metrics (T), object-oriented design metrics (D) and a combination of text and object-oriented design metrics (DT).

Our findings illustrate that using only text metrics does not perform as good as using only design-metrics (object-oriented design metrics). However, using text metrics in addition to design metrics leads to a small improvement over using only design metrics in the prediction of class inclusion/exclusion. On average, the addition of text metrics to object-oriented design metrics improved the prediction by 5.1%. Across different projects, the improvement of adding text-metrics to design metrics ranges from -6% to 22%. When taking the perspective of the classification algorithms, performance was improved using the combination features (DT) for all algorithms except k-NN (1) algorithm. By comparing the AUC score of all algorithms, the Random Forests produces the best result that indicates this algorithm is the most suitable algorithm for this purpose.

In terms of the predictive power of the text metrics, we found that the text metrics calculated from the individual case studies have better predictive power than the text metrics based on the combination of all case studies. This indicates that the text metrics features are truly domain-oriented.

This research demonstrates an improvement of class inclusion/exclusion prediction by using class names. We expect to further improve performance if we use other textual information such as operation-, parameter- and attribute-names as prediction features.

11.1.5 **RQ5:** *Does our automated framework for condensing of class diagrams help developers to understand the design of software systems?*

We validated our framework by conducting a semi-structured survey (a user study) to assess the respondents (students, academic researchers, IT professionals) opinion on the usefulness of the condensed class diagrams and our SAAbs tool in assisting in software comprehension.

The respondents were asked to give their opinion on a set of various class diagrams. We used the following diagrams: 1) the original forward design (FD); 2) the reverse engineered class diagram (RE-CD) (as produced by a CASE tool); and 3) abstracted RE-CDs (here 3 levels of abstraction were used: 25%, 50% and 75%). In total, 63 respondents participated this survey.

Our findings demonstrated that the level of 25% abstraction of RE-CD is rated most understandable compared to other diagrams. The results showed that the rate of understanding decreases when the amount of classes increases. We compared the respondents' preferred class diagrams between three levels of abstracted RE-CDs (25%, 50%, 75%) and the RE-CD (without abstraction).

The respondents were also asked to choose the one diagram that they prefer for using for system comprehension. The result shows that there are two main groups of the respondents: those that prefer to use the 25%-abstraction of the RE-CD and those that prefer the RE-CD. It is beyond our expectation that the RE-CD is chosen for system comprehension. Störrle [158] indicates that "layout quality does impact the understanding of UML diagram". When we observe the layout quality of the RE-CD (based on the four level design principle [159]), we found that the RE-CD presented in the questionnaire have a good layout quality, even though the number of classes is high. This may be the respondents' reason for choosing this diagram.

In this experiment, we also assess the respondents' judgment on our SAAbs tool that was developed to automate our approach. On average, the respondents give a score of 5.4 on a 6 point-scale for the usefulness of the tool. In the future, further studies should explore the use of the tool for performing maintenance tasks - both in experiments and in industrial settings.

11.2 Contributions

The contributions of this research are summarized as follows:

- **Discovery of developer reasoning about class diagram simplification.** This research found out which criteria developers use when selecting classes for creating a simplified system design.
- **Developing a classifier for class inclusion/exclusion prediction based on the object-oriented design features.** This research presented a novel approach for predicting class inclusion/exclusion using object-oriented design metrics as features. This approach showed how machine learning classification algorithms can be used to classify classes that could be included and classes that could be omitted.
- **Developing a classifier for class inclusion/exclusion based on the text features of class names.** We invented text metrics based on class names and enhanced the prediction performance by combining these text-based features with object-oriented design metrics.
- **An automated tool to support software comprehension by interactive exploration of various levels of design abstraction.** We developed an automated tool for scalable and interactive condensation of class diagrams. This tool provides multiple visualization techniques and offers various types of views to assist developers in software comprehension tasks.
- **Findings on the use of modeling in open source projects.** We expounded the use of UML modeling in open source projects. Amongst others, we described the types of diagrams used, the levels of detail of their representation and the frequency of updating models. In addition, we identified and explained a pattern regarding the change of focus on different types of diagram used over time. Also, we identified a relation between the size of the models and the size of the implementation.
- **Construction of a benchmark of open source projects using UML.** This research collected 10 open source projects that can be used as the benchmark for predicting important classes in class diagrams. The projects were derived from diverse types of domains and various sizes (the number of classes ranges from 59 to 900).

11.3 Discussion

In this section, we reflect on the result of this research.

11.3.1 Software Comprehension

To understand a system, software developers normally explore the system's artifacts (e.g. source code, software design). This activity supports the building of a cognitive design elements by software developers. Storey et al. [157][156] has proposed a set of 15 guidelines elements that are recommended for software exploration tools. Out of these 15 guidelines, we fulfilled 6 that fit the object-oriented system context.

In Chapter 5 and 6, we found that software developers focused on some information (such as class names and relationship) in class diagrams to understand a system. This is consistent with findings by Ko. et al [93] that software developers search for relevant code based on identifier-names and comments. Once the developers found the relevant code, they start to look at other related code (tracing relationships between classes).

In a broader perspective, the SAAbs tool provides multiple architecture views that may constitute an architecture reconstruction [96][139]. The work by Riva [144] identified multiple levels of reverse engineering (implementation, Design, Architecture). In contrast to our approach, those levels are seen as separate discrete levels which use concepts at different levels of abstraction for representing the system. We believe that extracting higher level concepts are a much needed step in reverse engineering. However, this requires a different ground truth compared to the one that we used in our research. An actual gap in abstraction levels needs to be bridged. Our approach focused instead on simplification through leaving out information.

11.3.2 Condensation of Class Diagrams

The condensation of class diagrams is the core discovery of this research. Commonly, condensation can be achieved in two ways: by abstraction or aggregation (or a combination thereof) [109]. In our research context, we use abstraction instead of aggregation because we want to facilitate both the bottom-up and top-down comprehension. For this purpose, aggregation is not suitable because this method sometimes presents too abstract views (or "big jump" view) compared to the complete design.

One may argue that using the abstraction method has the risk of loss precision and coherency (because of eliding details). Also, the question of "what is the right level of abstraction?" should be answered. We mitigated this risk and answered the aforementioned question by providing the multiple levels of design abstraction. In this way, the users may construct multiple levels of abstraction based on their need. Such scaling allows the gradual construction of the abstraction of from a big to a small number of classes (and vice versa). Also, the multiple levels of abstraction caters for the different demands on system views of different stakeholders.

In the following subsection, we discuss two important inputs of our abstraction methods: a) the "ground truth" and, b) the features used to identify the important classes.

The "Ground Truth"

To find the important classes in a class diagram, we apply a supervised machine learning approach. As the baseline or "ground truth", we use the classes that are included in the (man-made) forward design. We believed that the forward design is closely related to the system domain and represents the key system functionalities. Based on our survey in Chapter 5, most of the professional software developers indicated that the domain related class are important classes.

In contrast, Hammad et al. [77] and Bieman et al. [24] used version history information to establish their baseline of key classes. They assume that classes that frequently change in the evolution of a software are important classes. This is also a reasonable assumption. However, this approach also has some threats: classes that frequently change may also be classes that are not important in the domain, but more facility- or manager-classed such as lookup classes, log classes and graphical user interface (GUI) classes.

The Features to Identify the Important Classes

We used two main types of features for our classification: Object-oriented design metrics and text metrics. Both features proved to have some contribution to the ability to predict the important classes in a class diagram. We discuss these features in the following.

Feature: Object-oriented Design Metrics. Amongst the object-oriented design metrics, we only focused on metrics from the size- and coupling-category. As a result, we found that the Export Coupling Parameter (EC_Par), Dependency In (Dep_In) and NumOps are the most influential metrics for predicting the important classes.

The EC_Par and Dep_In metrics are coupling-type metrics. This means that the relationships of a class are an important factor in determining class importance. This result is aligned with the findings of Briand et al. [36] and Genero et al. [29] which also indicate that coupling is an important structural dimension in object-oriented design. Also, Steidl et al. [154] and Thung et al. [164] worked with information on class relationships to identify important classes in a class diagram. At the same time, coupling remains merely a syntactical aspects of a design.

From a maintenance perspective, error-prone classes can also be considered as important classes (because this is where lots of maintenance effort will be focussed). Work in the area of prediction of faulty classes by Zhou et al. [188] and Gyimothy et al. [73] mentions that coupling and numbers of methods (WMC) are the most influential features to predict faulty classes. Likewise, we also found that the Number of operations (same with WMC) is one of the influential features. Hence, we can conclude that the object-oriented design metrics are useful generic, application-independent features for predicting important classes.

Feature: Text Metrics. This research also shows that text metrics based on class names could be used as predictors for class importance. However, the object-oriented design metrics perform better than the text metrics. Our prediction's performance increases only when we combine these two predictors sets together. Work related to code summarization [74][75][50] indicates that method-names and class-names are suitable to summarize a class. These works found that method-names constitute better criteria to summarize a class compared to class-names. Based on this, we expect a better performance of our approach if we include other textual elements such as text in methods, parameters and attributes.

The text-metrics that are calculated based on individual software projects (case studies) demonstrated a higher predictive power compared to metrics calculated based on the combined set of software projects. The projects in our study come from different types of domain; hence, it is difficult to retrieve or predict the domain-related words across all domains. Klint et al. [91] shows that the collection of domain-related words are challenging and requires a lot of efforts for a particular domain; thus, a text metrics-based generalize the prediction model is difficult to realize.

The research opens an extensive perspective of condensing class diagrams in order to ease the system comprehension. In the next section, we discuss directions for future work.

11.4 Future Work

Our research can be considered as an initial work on simplifying reverse engineered UML diagrams through machine learning. Based on the respondents of our surveys (Chapter 5, 6 and 10), we can outline several directions for future work to improve this work as well as implementing the technique to a broader perspective. The remaining subsections describe directions for future work.

11.4.1 Enriching the Ground Truth

In Chapter 7 and 8, we used supervised machine learning to classify class inclusion/exclusion. We assume that forward designed class diagrams (i.e. provided as part of the documentation of these projects we studied) as the 'ground truth'. We believe that incorporating other information as the 'ground truth' may improve the classification result. Several suggestions on which complementary sources of information to use are the following:

- **Version history:** Classes that frequently change during the software evolution could be classes that are important to the system. Also, other information such as the severity level (based on defect classifications for a particular class) could be a candidate to enrich the 'ground truth'.

- **Eye tracking information:** One small field of empirical research focuses on detecting important information in design documents based on tracking the focus of human eyes when looking at a software design on a computer screen. The information that users focus on in class diagrams could be additional information to our 'ground truth'.

- **Software documentation:** We only used the classes that are presented in the class diagram (in software documentation) for the 'ground truth'. We believe there are more information in the text (such as classes that frequently mentioned) that may improve our baseline of this study.

- **Interactive Learning:** Thung et al. [164] have demonstrated the improvement of the AUC score when an optimistic classification technique is used. This technique is a form of semi-supervised learning technique where users can give input to generate a finer statistical model. Therefore, we believe that interactive and dynamic refinement of the 'ground truth' may improve the selection of the important classes in class diagrams.

- **Dynamic Analysis:** Dynamic analysis is the analysis of the properties of a running program [19]. This analysis can for example, measure the classes that are being used most by an application. This information can be used to enhanced the 'ground truth'.

11.4.2 Exploring Features

This research has looked at the use of object-oriented design metrics, text metrics and network metrics (work by Thung et al. [164]) as features for classifying class inclusion/exclusion. We believe that there is other information that may be used as features to predict class inclusion/exclusion. We believe that information from operation-, parameter- and attribute-names may offer a significant predictive power to the classification. There is also another possibility to use the information from the software repository as additional features. Information such as source code metrics and text from bug reports may be added to the features to improve the classification result.

11.4.3 Task-oriented Validation

Preliminary findings of our user study show that the result of our approach and tool is understandable and helpful for the software comprehension task. Therefore, we believe that a validation based on more realistic maintenance tasks is required to strengthen the validation of our proposed framework and get a broader perspective to enhance the tool and technique. Through such a task-oriented validation, one could gather more information on the tool's actual usage.

11.4.4 Class Segmentation

The respondents in Chapter 5 suggested a segmentation (grouping) of classes based on features and functionality: Classes that have similar functionality or together implement one feature should preferably be presented in the same segment or group. Novel feature location techniques are needed to identify features in the class diagrams.

Another possible technique that can be used for class segmentation is program slicing [169],[181]. Program slicing removes those parts of the program that have no effect upon the semantic interest and concentrates on the selected aspects related to some concern. Often these techniques are based on tracing of data-flow and dependency analysis. It is also possible to combine the aforementioned technique with our approach. Provided with the information of a specific feature (by using program slicing or feature location), our approach can produce a rank of classes that suggest the relevant classes to the features. Our existing visualization feature may help to visualize the segmentation (and its embedding in a larger design).

11.4.5 Visualization of Result

In Chapter 10, the respondents gave several valuable suggestions to improve the presentation of the class diagram viewer. Therefore, we believe the following enhancement may benefit to the tool:

- *Layout:* Investigation and application of more layout options such as central and hierarchical layout.
- *Interactive Viewer:* Interactive editing to allow users full control of the layout of diagrams.

Appendix A

Case Study Candidates

*In this Appendix, we list the candidate projects for our case studies (Table A.1). Item mark with * are the selected case studies.*

Table A.1: *List of Case Study Candidates*

No	Project	Source
1	**ArgoUML***	**http://sourceforge.net/projects/argouml/**
2	Bookszenbooks	http://code.google.com/p/bookszenbooks/
3	cmpt371t1	http://code.google.com/p/cmpt371t1/
4	Concurrentadt	http://code.google.com/p/concurrentadt/
5	CrimsonPortal	http://code.google.com/p/crimsonportal/
6	DBForms	http://jdbforms.sourceforge.net/UsersGuide/html /index.html
7	DocDoku	http://code.google.com/p/docdoku/
8	driving-bc	http://code.google.com/p/driving-bc/
9	DRMJ-Webshop	http://code.google.com/p/drmj-webshop/
10	EmpScheduler2008	http://code.google.com/p/empscheduler2008/
11	Epydoc	http://epydoc.sourceforge.net/api/epydoc-module.html
12	europa-ps	http://code.google.com/p/europa-pso/
13	Fipa-english-auction	http://code.google.com/p/fipa-english-auction-interaction-protocol/
14	Fuge	http://fuge.sourceforge.net/dev/index.php
15	gelsanalyzer	http://code.google.com/p/gelsanalyzer/downloads /detail?name=MVC%20Class%20Diagram %20Iteration2.png&can=2&q=

16	GNetWatch	http://gnetwatch.sourceforge.net/doc.html
17	Google-voice-java	http://code.google.com/p/google-voice-java/
18	Gotogate	http://gotogate.googlecode.com/svn/trunk/
19	gwavmerger	http://gwavmerger.sourceforge.net/gwm-design/design.html
20	**gwt-portlets***	**http://code.google.com/p/gwtportlets/**
21	gwtuml	http://code.google.com/p/gwtuml/
22	httpdbase4j	http://httpdbase4j.berlios.de/
23	Jalli	http://jalli.berlios.de/
24	Java Kohonen Neural Network Library	http://jknnl.sourceforge.net/
25	Javaassessment	http://code.google.com/p/javaassessment /downloads/list
26	**JavaClient***	**http://java-player.sourceforge.net/**
27	**JGAP***	**https://sourceforge.net/project/screenshots.php?group_id=11618**
28	jjack	http://jjack.berlios.de/
29	**jpmc***	**http://jpmc.sourceforge.net/diagrams.html**
30	Jvending	http://jvending.sourceforge.net/
31	Krank	http://code.google.com/p/krank/
32	Mandragora Project	http://mandragora.sourceforge.net/referenceguide/how-to-extend.html
33	**Mars Simulation***	**https://sourceforge.net/apps/mediawiki/mars-sim/index.php?title=UML_Diagrams**
34	**Maze-Solver***	**http://code.google.com/p/maze-solver/wiki/MazeModelDoc**
35	monopolj	http://code.google.com/p/monopolj/
36	MyDas	http://code.google.com/p/mydas/
37	Netbeams	http://code.google.com/p/netbeams/wiki/ DataSensorPlatform
38	network-keeper	http://code.google.com/p/network-keeper/wiki/Diagrams
39	**Neuroph***	**http://neuroph.sourceforge.net/**
40	nmt-cs326-g5	https://code.google.com/p/nmt-cs326-g5/
41	Novosoft Metadata Framework	http://nsuml.sourceforge.net/index.html
42	Nymp	http://code.google.com/p/nymph/
43	ObjectLabKit	http://objectlabkit.sourceforge.net/apidocs /net/objectlab/kit/datecalc/jdk/CalendarPeriodCountCalculator.html

44	ObjectListView	http://objectlistview.sourceforge.net/cs/index.html
45	OpenMeeting	http://code.google.com/p/openmeetings/
46	pacpounder	https://code.google.com/p/pacpounder/
47	Pendulim	http://code.google.com/p/pendulim/
48	Portions	http://portions.sourceforge.net/en/guide.html
49	Primitive Collections	http://pcj.sourceforge.net/docs/guide/pcj-guide.html
50	Qt OIC Container 3.5	http://qtioccontainer.sourceforge.net/uml.html
51	RandyLoops	http://randyloops.sourceforge.net/
52	rpcstruts	https://code.google.com/p/rcpstruts/
53	StarUML	http://staruml.sourceforge.net/docs/api-doc/Modeling%20Elements/UML%20Model %20Elements /Foundation/Core/package-summary.html
54	tiOPF	http://tiopf.sourceforge.net/Doc/Concepts /4_BuildingAnAbstractBOMWithTheComposite.shtml
55	Topology data scripting	https://sourceforge.net/apps/mediawiki/free-cad/index.php?title=Topological_data_scripting
56	**wro4J***	**http://code.google.com/p/wro4j/**
57	**Xuml-compiler***	**http://code.google.com/p/xuml-compiler/**

List of Figures

List of Tables

Bibliography

[1] IEEE standard for software maintenance. *IEEE Std 1219-1998*, pages i–, 1998. (cited on page 15).

[2] BerliOS. `http://www.berlios.de/`, 2012. (cited on page 33).

[3] Google search engine. `http://www.google.com`, 2012. (cited on page 33).

[4] GoogleCode. `http://code.google.com/`, 2012. (cited on page 33).

[5] Limeservice. `https://www.limeservice.com/en/`, 2012. (cited on pages 95 and 98).

[6] Object-oriented dataset. `http://www.liacs.nl/~hosman/DataSets.rar`, 2012. (cited on pages 6 and 123).

[7] SourceForge. `http://sourceforge.net/`, 2012. (cited on page 33).

[8] GitHub. `https://github.com/`, 2014. (cited on page 33).

[9] MagicDraw. `http://www.nomagic.com/`, 2014. (cited on pages 34, 121 and 139).

[10] PlantUML. `http://plantuml.sourceforge.net/`, 2014. (cited on page 165).

[11] RapidMiner. `http://rapidminer.com/`, 2014. (cited on pages 140 and 165).

[12] A. Abraham. Rule-based expert systems. *Handbook of measuring system design*, 2005. (cited on page 22).

[13] D. Akehurst, G. Howells, and K.M. Maier. Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling*, 6(1):3–35, March 2007. (cited on pages 49 and 64).

[14] J. Al Dallal and L.C. Briand. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Trans. Softw. Eng. Methodol.*, 21(2):8:1–8:34, March 2012. (cited on page 28).

[15] S.A. Alvarez. An exact analytical relation among recall, precision, and classification accuracy in information retrieval. *Boston College, Boston, Technical Report BCCS-02-01*, pages 1–22, 2002. (cited on page 27).

[16] J.R. Anderson, R.S. Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine learning: An artificial intelligence approach*, volume 2. Morgan Kaufmann, 1986. (cited on page 22).

[17] O. Andriyevska, N. Dragan, B. Simoes, and J.I. Maletic. Evaluating UML class diagram layout based on architectural importance. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '05, pages 1–6, Washington, DC, USA, 2005. IEEE Computer Society. (cited on page 43).

[18] R.A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. (cited on page 139).

[19] T. Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999. (cited on pages 17 and 198).

[20] V. R. Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3 – 12, 1997. (cited on page 11).

[21] S. Bassil and R.K. Keller. Software visualization tools: survey and analysis. In *Proceedings of 9th International Workshop on Program Comprehension (IWPC 2001)*, pages 7–17, 2001. (cited on page 71).

[22] B. Bellay and H. Gall. A comparison of four reverse engineering tools. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, WCRE '97, pages 2–, Washington, DC, USA, 1997. IEEE Computer Society. (cited on pages 49, 70 and 118).

[23] K.H. Bennett and V.T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM. (cited on page 11).

[24] J.M. Bieman, A.A. Andrews, and H.J. Yang. Understanding change-proneness in OO software through visualization. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 44–53, 2003. (cited on pages 158 and 196).

[25] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster. Program understanding and the concept assignment problem. *Communication ACM*, 37(5):72–82, May 1994. (cited on page 11).

[26] D. Billsus and M.J. Pazzani. Learning collaborative information filters. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 46–54. Morgan Kaufmann Publishers Inc., 1998. (cited on page 27).

[27] A.B. Binkley and S.R. Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the 20th international conference on Software engineering*, pages 452–455. IEEE Computer Society, 1998. (cited on page 93).

[28] R.C. Bjork. ATM simulation system. `http://www.math-cs.gordon.edu/courses/cs211/ATMExample/`, 2002. (cited on pages 52, 96 and 175).

[29] M.G. Bocco, M. Piattini, and C. Calero. A survey of metrics for UML class diagrams. *Journal of Object Technology*, 4(9):59–92, 2005. (cited on pages 121 and 196).

[30] A. Boklund, S. MankeFors-Christiernin, C. Johansson, and H. Lindell. A comparative study of forward and reverse engineering in UML tools. IADIS International Conference Applied Computing 2007, 2007. (cited on page 49).

[31] G. Booch. *Object Solutions: Managing the Object-oriented Project*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995. (cited on page 18).

[32] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999. (cited on page 31).

[33] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. (cited on page 26).

[34] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for C++. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 412–421, New York, NY, USA, 1997. ACM. (cited on pages 122 and 162).

[35] L.C. Briand, J. Wüst, J.W. Daly, and D. Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3):245–273, 2000. (cited on page 93).

[36] L.C. Briand, J. Wüst, S.V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: An industrial case study. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 345–354, New York, NY, USA, 1999. ACM. (cited on pages 93, 121 and 196).

[37] R. Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983. (cited on page 13).

[38] O. Chapelle, B. Schlkopf, and A. Zien. *Semi-Supervised Learning*. The MIT Press, 1st edition, 2010. (cited on page 23).

[39] N.V. Chawla, K.W. Bowyer, L.O. Hall, and W.P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002. (cited on page 28).

[40] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, June 1994. (cited on pages 122 and 162).

[41] E.J. Chikofsky and J.H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, January 1990. (cited on pages 16, 17, 92, 118 and 205).

[42] E. Chung, C. Jensen, K. Yatani, V. Kuechler, and K.N. Truong. Sketching and drawing in the design of open source software. In *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '10, pages 195–202, Washington, DC, USA, 2010. IEEE Computer Society. (cited on page 33).

[43] C.L. Corritore and S. Wiedenbeck. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies*, 54(1):1–23, 2001. (cited on pages 12 and 13).

[44] A. Craig, A. Dinardo, and R. Gillespie. Pacman game. `http://code.google.com/p/tb-pacman/`, 2009. (cited on pages 72, 97 and 176).

[45] P. Dayan, M. Sahani, and G. Deback. Unsupervised learning. In *The MIT Encyclopedia of the Cognitive Sciences*. The MIT Press, 1999. (cited on page 23).

[46] S. Demeyer. Research methods in computer science. In *Procedings of the International Conference of Software Maintenance (ICSM 2011)*, page 600, 2011. (cited on page 5).

[47] B. Dobing and J. Parsons. Current practices in the use of UML. In *ER (Workshops)*, volume 3770 of *Lecture Notes in Computer Science*, pages 2–11. Springer, 2005. (cited on pages 32 and 33).

[48] B. Dobing and J. Parsons. How UML is used. *Commun. ACM*, 49(5):109–113, May 2006. (cited on pages 32 and 33).

[49] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Software Eng.*, 35(4):573–591, 2009. (cited on page 13).

[50] B.P. Eddy, J.A Robinson, N.A Kraft, and J.C. Carver. Evaluating source code summarization techniques: Replication and expansion. In *Proceedings of the IEEE 21st International Conference on Program Comprehension (ICPC)*, pages 13–22, May 2013. (cited on pages 137 and 197).

[51] A. Egyed. Semantic abstraction rules for class diagrams. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 301–304. IEEE, 2000. (cited on page 93).

[52] A. Egyed. Automated abstraction of class diagrams. *ACM Trans. Softw. Eng. Methodol.*, 11(4):449–491, October 2002. (cited on pages 93 and 131).

[53] K. El Emam, W. Melo, and J.C. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1):63–75, 2001. (cited on page 93).

[54] J. Erickson and K. Siau. Theoretical and practical complexity of modeling methods. *Commun. ACM*, 50(8):46–51, August 2007. (cited on page 33).

[55] H.-E. Eriksson, M. Penker, and D. Fado. *UML 2 Toolkit*. John Wiley & Sons, Inc., New York, NY, USA, 2003. (cited on pages 73 and 96).

[56] J.-R. Falleri, M. Huchard, and C. Nebut. A generic approach for class model normalization. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 431–434, Washington, DC, USA, 2008. IEEE Computer Society. (cited on page 93).

[57] J.-M. Favre. GSEE: a generic software exploration environment. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*, pages 233–244, 2001. (cited on page 13).

[58] T. Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006. (cited on pages 25, 27 and 28).

[59] A.M. Fernández-Sáez, M.R.V. Chaudron, M. Genero, and I. Ramos. Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: A controlled experiment. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, pages 60–71, New York, NY, USA, 2013. ACM. (cited on pages 4, 92, 118 and 156).

[60] A.M. Fernández-Sáez, M. Genero, and M.R.V. Chaudron. Does the level of detail of UML models affect the maintainability of source code? In *Proceedings of the 2011th International Conference on Models in Software Engineering*, MODELS'11, pages 134–148, Berlin, Heidelberg, 2012. Springer-Verlag. (cited on page 36).

[61] A. Field. *Discovering Statistics Using SPSS*. SAGE Publications, 2005. (cited on page 26).

[62] G. Forman and M. Scholz. Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement. *ACM SIGKDD Explorations Newsletter*, 12(1):49–57, 2010. (cited on pages 28 and 126).

[63] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003. (cited on pages 37, 38 and 52).

[64] A.K. Gahalut and P. Khandnor. Reverse engineering : An essence for software re-engineering and program analysis. *International Journal of Engineering and Technology*, 2:2296–2303, 2010. (cited on page 49).

[65] M. Grechanik, K.S. McKinley, and D.E. Perry. Recovering and using Use-case-diagram-to-source-code traceability links. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 95–104, New York, NY, USA, 2007. ACM. (cited on page 37).

[66] T.J. Grose, G.C. Doney, and S.A. Brodsky. *Mastering XMI: Java Programming with XMI, XML and UML*. John Wiley & Sons, Inc., New York, NY, USA, 2001. (cited on page 21).

[67] M. Grossman, J.E. Aronson, and R.V. McCarthy. Does UML make the grade? Insights from the software development community. *Inf. Softw. Technol.*, 47(6):383–397, April 2005. (cited on page 33).

[68] Object Management Group. Unified Modeling Language (UML), Superstructure, Version 2.4.1, August 2011. (cited on pages 18, 39, 52 and 205).

[69] Object Management Group. ISO/IEC 19509:2014 Information technology – Object management group XML metadata interchange (XMI). `http://www.iso.org/iso/catalogue_detail.htm?csnumber=61845`, 2014. (cited on page 21).

[70] Y.-G. Guéhéneuc. A systematic study of UML class diagram constituents for their abstract and precise recovery. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 265–274, Washington, DC, USA, 2004. IEEE Computer Society. (cited on page 92).

[71] Y.-G Guéhéneuc. Taupe: towards understanding program comprehension. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 1. IBM Corp., 2006. (cited on page 21).

[72] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 301–314, New York, NY, USA, 2004. ACM. (cited on pages 48, 54, 57 and 59).

[73] T. Gyímothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005. (cited on pages 93, 121 and 196).

[74] S. Haiduc, J. Aponte, and A Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, volume 2, pages 223–226, May 2010. (cited on pages 136 and 197).

[75] S. Haiduc, J. Aponte, L. Moreno, and A Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 35–44, Oct 2010. (cited on pages 137 and 197).

[76] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: An update, 2009. (cited on pages 26, 127, 146, 147 and 165).

[77] M. Hammad, M.L. Collard, and J.I. Maletic. Measuring class importance in the context of design evolution. In *Proceedings of the IEEE 18th International Conference on Program Comprehension (ICPC 2010)*, pages 148–151, 2010. (cited on pages 120, 133, 157 and 196).

[78] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011. (cited on page 154).

[79] J.A. Hanley and B.J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143(1):29–36, April 1982. (cited on pages 28 and 147).

[80] T. Hettel, M. Lawley, and K. Raymond. Model synchronisation: Definitions for round-trip engineering. In *Theory and Practice of Model Transformations*, pages 31–45. Springer, 2008. (cited on page 48).

[81] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM. (cited on page 33).

[82] M. Ichii, T. Myojin, Y. Nakagawa, M. Chikahisa, and H. Ogawa. A Rule-based automated approach for extracting models from source code. In *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE '12, pages 308–317, Washington, DC, USA, 2012. IEEE Computer Society. (cited on page 43).

[83] I. Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, 1 edition, June 1992. (cited on page 18).

[84] G. Jalloul. *UML by Example*. Cambridge University Press, New York, NY, USA, 2004. (cited on page 19).

[85] S. Jarzabek. *Effective software maintenance and evolution: A reuse-based approach*. CRC Press, 2007. (cited on page 17).

[86] R. Kazman and S.J. Carriere. View extraction and view fusion in architectural understanding. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 290–299, Jun 1998. (cited on page 13).

[87] S. Kearney and J.F. Power. REM4j - A framework for measuring the reverse engineering capability of UML CASE tools. In *SEKE*, pages 209–214. Knowledge Systems Institute Graduate School, 2007. (cited on page 50).

[88] A. Kent and J.G. Williams. *Encyclopedia of Computer Science and Technology: Volume 35 - Supplement 20: Acquiring Task-Based Knowledge and Specifications to Seek Time Evaluation*. Encyclopedia of Computer Science Series. Taylor & Francis, 1996. (cited on page 12).

[89] A. Kent and J.G. Williams. *Encyclopedia of Microcomputers: Volume 25 - Supplement 4*. Microcomputers Encyclopedia. Taylor & Francis, 2000. (cited on page 12).

[90] B. Klatt, M. Küster, K. Krogmann, and O. Burkhardt. A change impact analysis case study: Replacing the input data model of SoMoX. *Recall*, 71:99–58. (cited on page 158).

[91] P. Klint, D. Landman, and J. Vinju. Exploring the limits of domain model recovery. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM2013)*, pages 120–129, Sept 2013. (cited on page 197).

[92] A.J. Ko and B.A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1):41–84, 2005. (cited on page 4).

[93] A.J. Ko, B.A Myers, M.J. Coblenz, and H.H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, Dec 2006. (cited on pages 4 and 195).

[94] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 22–, Washington, DC, USA, 2002. IEEE Computer Society. (cited on pages 49 and 64).

[95] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *Journal of Software Maintenance*, 15(2):87–109, March 2003. (cited on page 71).

[96] R. Koschke. Architecture reconstruction. In Andrea De Lucia and Filomena Ferrucci, editors, *Software Engineering*, volume 5413 of *Lecture Notes in Computer Science*, pages 140–173. Springer Berlin Heidelberg, 2009. (cited on page 195).

[97] J. Koskinen and T. Lehmonen. Analysis of ten reverse engineering tools. In K. Elleithy, editor, *Advanced Techniques in Computing Sciences and Software Engineering*, pages 389–394. Springer Netherlands, 2010. (cited on pages 48 and 49).

[98] S.B. Kotsiantis. Supervised machine learning: A review of classification techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press. (cited on pages 23, 24 and 205).

[99] A. Lake and C.R. Cook. Use of factor analysis to develop OOP software complexity metrics. Technical report, Oregon State University, Corvallis, OR, USA, 1994. (cited on pages 122 and 162).

[100] M. Lanza and S. Ducasse. Polymetric views - A lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept 2003. (cited on page 13).

[101] J.-F Le Gall. Random trees and applications. *Probab. Surv*, 2(245-311):15–43, 2005. (cited on page 26).

[102] F. Leemhuis. Devnology community. `http://devnology.nl/`, 2012. (cited on page 74).

[103] T.C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Softw.*, 20(6):35–39, November 2003. (cited on pages 3 and 48).

[104] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993. (cited on page 93).

[105] D. Lin and P. Pantel. Induction of semantic classes from natural language text. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 317–322, New York, NY, USA, 2001. ACM. (cited on page 138).

[106] J.B. Lovins. Development of a Stemming Algorithm. June 1968. (cited on page 141).

[107] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):31, 2014. (cited on page 12).

[108] S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. *International Conference on Program Comprehension*, 1998. (cited on page 158).

[109] M.T. Maybury. Generating summaries from event data. *Inf. Process. Manage.*, 31(5):735–751, September 1995. (cited on page 195).

[110] A. McCallum and K. Nigam. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998. (cited on page 26).

[111] S. Medini, G. Antoniol, Y.-G. Guéhéneuc, M. Di Penta, and P. Tonella. SCAN: An approach to label and relate execution trace segments. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE 2012)*, pages 135–144, Oct 2012. (cited on page 137).

[112] G.A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, (2):81–97, March 1956. (cited on page 4).

[113] T.M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. (cited on pages 22 and 23).

[114] L. Moreno, J. Aponte, G. Sridhara, A Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the IEEE 21st International Conference on Program Comprehension (ICPC)*, pages 23–32, May 2013. (cited on page 137).

[115] NetBeans. Netbeans Integrated Development Environment (IDE). `https://netbeans.org/features/index.html`, 2014. (cited on page 165).

[116] D. Ng, D.R. Kaeli, S. Kojarski, and D.H. Lorenz. Program comprehension using aspects. In *ICSE 2004 Workshop WoDiSEE'2004*, 2004. (cited on page 12).

[117] A. Nugroho. *The effects of UML modeling on the quality of software*. PhD thesis, Leiden University, the Netherlands, 2010. (cited on page 73).

[118] A. Nugroho and M.R.V. Chaudron. A survey of the practice of design – Code correspondence amongst professional software engineers. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07, pages 467–469, Washington, DC, USA, 2007. IEEE Computer Society. (cited on pages 48, 70 and 118).

[119] Object Management Group (OMG). `http://www.omg.org`, 2014. (cited on page 32).

[120] M. Orr. Introduction to radial basis function networks. Technical report, Institute for Adaptive and Neural Computation, Edinburgh University, 1996. (cited on page 26).

[121] M.H. Osman. Answered questionnaire. `http://www.liacs.nl/~hosman/ValidationAll.rar`, 2014. (cited on pages 177 and 186).

[122] M.H. Osman. An example of the text dictionary. `www.liacs.nl/~hosman/TextDictionary.csv`, 2014. (cited on pages 6 and 141).

[123] M.H. Osman. The list of common and uncommon words. `http://www.liacs.nl/~hosman/CommonAndUncommon.xlsx`, 2014. (cited on page 142).

[124] M.H. Osman. SAAbs tool demonstration. http://www.youtube.com/watch?v=dHBB5wA2wDI, 2014. (cited on pages 152 and 165).

[125] M.H. Osman and M.R.V. Chaudron. Correctness and completeness of case tools in reverse engineering source code into uml model. *GSTF Journal on Computing*, 2(1):193–201, 2012. (cited on pages 36 and 118).

[126] M.H. Osman and M.R.V. Chaudron. An assessment of reverse engineering capabilities of UML CASE tools. In *Proceedings of 2nd Annual International Conference on Software Engineering and Application (SEA 2011)*, pages 7–12, December 12-13, 2011. (cited on page 92).

[127] M.H. Osman, M.R.V. Chaudron, and P. van der Putten. An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM 2013)*, pages 140–149, Sept 2013. (cited on pages 157, 163 and 187).

[128] M.H. Osman, M.R.V. Chaudron, and P. van der Putten. Condensing reverse engineered class diagram using text mining. Technical report, Leiden University, the Netherlands (`http://www.liacs.nl/~hosman/TechnicalReport-2014-02.pdf`), 2014. (cited on pages 162 and 163).

[129] M.H. Osman and A. van Zadelhoff. Original questionnaire. `http://www.liacs.nl/~hosman/Questionnaire.rar`, 2012. (cited on page 72).

[130] M.H. Osman and A. van Zadelhoff. Structured questionnaire. `http://www.liacs.nl/~hosman/The_Presence_of_Classes_in_Class_Diagrams.pdf`, 2012. (cited on page 95).

[131] M.H. Osman and A. van Zadelhoff. Survey data. `http://www.liacs.nl/~hosman/SurveyData.rar`, 2012. (cited on page 74).

[132] M.H. Osman, A. van Zadelhoff, and M.R.V. Chaudron. UML class diagram simplification - A survey for improving reverse engineered class diagram comprehension. In *MODELSWARD*, pages 291–296, 2013. (cited on page 160).

[133] M.H. Osman, A. van Zadelhoff, D.R. Stikkolorum, and M.R.V. Chaudron. UML class diagram simplification: What is in the developer's mind? In *Proceedings of the Second Edition of the International Workshop on Experiences and Empirical Studies in Software Modelling*, EESSMod '12, pages 5:1–5:6, New York, NY, USA, 2012. ACM. (cited on pages 43, 160 and 165).

[134] M.H. Osman and L. Wei. The SAAbs tool. https://github.com/aislimau/SAAbs, 2014. (cited on pages 6 and 165).

[135] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987. (cited on page 12).

[136] F. Perin, L. Renggli, and J. Ressia. Ranking software artifacts. In *4th Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2010)*, 2010. (cited on pages 120 and 133).

[137] S.L. Pfleeger and B.A. Kitchenham. Principles of survey research: Part 1: Turning lemons into lemonade. *SIGSOFT Softw. Eng. Notes*, 26(6):16–18, November 2001. (cited on page 5).

[138] H. Pirzadeh, A. Hamou-Lhadj, and M. Shah. Exploiting text mining techniques in the analysis of execution traces. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*, pages 223–232, Sept 2011. (cited on page 137).

[139] D. Pollet, S. Ducasse, L. Poyet, I Alloui, S. Cimpan, and H. Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR '07)*, pages 137–148, March 2007. (cited on page 195).

[140] M.F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. (cited on page 141).

[141] F.J. Provost, T. Fawcett, and R. Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 445–453, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. (cited on page 154).

[142] F.D. Rácz and K. Koskimies. Tool-supported compression of UML class diagrams. In *«UML»'99—The Unified Modeling Language*, pages 172–187. Springer, 1999. (cited on page 4).

[143] A. Rajaraman and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011. (cited on page 140).

[144] C. Riva. Reverse architecting: an industrial experience report. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE2000)*, pages 42–50, 2000. (cited on page 195).

[145] S. Robitaille, R. Schauer, and R.K. Keller. Bridging program comprehension tools by design navigation. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2000)*, pages 22–32, 2000. (cited on page 13).

[146] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. (cited on page 18).

[147] J. Rumbaugh, I. Jacobson, and G. Booch, editors. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999. (cited on page 19).

[148] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, April 2009. (cited on page 5).

[149] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:211–229, 1959. (cited on page 23).

[150] T. Schäfer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT software exploration tool. *IEEE Trans. Softw. Eng.*, 32(9):753–768, 2006. (cited on page 13).

[151] B. Sharif and J.I. Maletic. The effect of layout on the comprehension of UML class diagrams: A controlled experiment. In *5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2009)*, pages 11–18. IEEE, 2009. (cited on page 21).

[152] B. Shneiderman and R.E. Mayer. Interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8(3):219–238, 1979. (cited on page 12).

[153] SparxSystems. Enterprise Architect. `http://www.sparxsystems.com.au/`, 2013. (cited on pages 34, 72 and 94).

[154] D. Steidl, B. Hummel, and E. Juergens. Using network analysis for recommendation of central software classes. In *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE '12, pages 93–102, Washington, DC, USA, 2012. IEEE Computer Society. (cited on pages 120, 133, 157 and 196).

[155] P. Stevens. Small-scale XMI programming: A revolution in UML tool use? *Automated Software Eng.*, 10(1):7–21, January 2003. (cited on pages 22 and 159).

[156] M.-A.D. Storey, F.D. Fracchia, and H.A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171 – 185, 1999. (cited on pages 15, 156, 195 and 205).

[157] M.-A.D. Storey, F.D. Fracchia, and H.A. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, WPC '97, pages 17–, Washington, DC, USA, 1997. IEEE Computer Society. (cited on pages 14, 132, 168 and 195).

[158] H. Störrle. On the impact of layout quality to understanding UML diagrams. In *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 135–142. IEEE, 2011. (cited on pages 21 and 193).

[159] H. Störrle. On the impact of layout quality to understanding UML diagrams: Size matters. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 518–534. Springer International Publishing, 2014. (cited on pages 21 and 193).

[160] D. Sun and K. Wong. On evaluating the layout of UML class diagrams for program comprehension. In *Proceedings of 13th International Workshop on Program Comprehension (IWPC 2005)*, pages 317–326. IEEE, 2005. (cited on page 21).

[161] R.S. Sutton. Introduction: The challenge of reinforcement learning. *Machine Learning*, 8(3-4):225–227, 1992. (cited on page 24).

[162] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, Finland, 2000. (cited on page 4).

[163] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 242–249. IEEE, 1999. (cited on page 93).

[164] F. Thung, D. Lo, M.H. Osman, and M.R.V. Chaudron. Condensing class diagrams by analyzing design and network metrics using optimistic classification. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, pages 110–121, New York, NY, USA, 2014. ACM. (cited on pages 157, 187, 196 and 198).

[165] Tigris.org. ArgoUML. `http://argouml.tigris.org/`, 2014. (cited on page 39).

[166] S. Tilley and S. Huang. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *Proceedings of the 21st annual international conference on Documentation*, pages 184–191. ACM, 2003. (cited on page 21).

[167] S.R. Tilley, P. Santanu, and D.B. Smith. Towards a framework for program understanding. In *Proceedings of 4th Workshop on Program Comprehension (WPC'96)*, pages 19–28. IEEE, 1996. (cited on page 168).

[168] K.M. Ting. Matching model versus single model: A study of the requirement to match class distribution using decision trees. In J.-F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, editors, *Machine Learning: ECML 2004*, volume 3201 of *Lecture Notes in Computer Science*, pages 429–440. Springer Berlin Heidelberg, 2004. (cited on page 28).

[169] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995. (cited on page 199).

[170] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä. Empirical studies in reverse engineering: State of the art and future trends. *Empirical Softw. Engg.*, 12(5):551–571, October 2007. (cited on page 118).

[171] P. van der Putten and M. van Someren. A Bias-variance analysis of a real world learning problem: The CoIL challenge 2000. *Mach. Learn.*, 57(1-2):177–195, October 2004. (cited on pages 25 and 26).

[172] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings of the 4th IEEE/IFIP Working Conference on Software Architecture (WICSA 2004)*, pages 122–132. IEEE, 2004. (cited on pages 3 and 70).

[173] A. von Mayrhauser and A.M. Vans. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering (CASE '93)*, pages 230–239, Jul 1993. (cited on page 13).

[174] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995. (cited on pages 13, 14 and 205).

[175] E. Voorhees. Natural language processing and information retrieval. In M. Pazienza, editor, *Information Extraction*, volume 1714 of *Lecture Notes in Computer Science*, pages 32–48. Springer Berlin Heidelberg, 1999. (cited on page 139).

[176] W3C. Extensible markup language (XML). `http://www.w3.org/XML/`, 2014. (cited on page 21).

[177] B.E. Wampler. *The Essence of Object-Oriented Programming with Java and UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. (cited on pages 52 and 57).

[178] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Amsterdam, 3 edition, 2011. (cited on pages 23, 25, 26 and 121).

[179] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G.J. McLachlan, A. Ng, B. Liu, P.S. Yu, Z.-H. Zhou, M. Steinbach, D.J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, December 2007. (cited on page 26).

[180] J. Wüst. SDMetrics. `http://www.sdmetrics.com/`, 2013. (cited on pages 35, 50, 94, 95, 121, 159 and 209).

[181] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005. (cited on page 199).

[182] K. Yatani, E. Chung, C. Jensen, and K.N. Truong. Understanding how and why open source contributors use diagrams in the development of Ubuntu. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 995–1004, New York, NY, USA, 2009. ACM. (cited on page 33).

[183] S. Yusuf, H. Kagdi, and J.I. Maletic. Assessing the comprehension of UML class diagrams via eye tracking. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 113–122, Washington, DC, USA, 2007. IEEE Computer Society. (cited on pages 21 and 70).

[184] A. Zaidman and S. Demeyer. Automatic identification of key classes in a software system using webmining techniques. *J. Softw. Maint. Evol.*, 20(6):387–417, November 2008. (cited on pages 119 and 133).

[185] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, May 1998. (cited on page 6).

[186] W. Zhang, Y. Yang, and Q. Wang. Network analysis of OSS evolution: An empirical study on ArgoUML project. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 71–80, New York, NY, USA, 2011. ACM. (cited on page 42).

[187] J. Zhi, V. Garousi-Yusifoğlu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99(0):175 – 198, 2015. (cited on page 32).

[188] Y. Zhou and H. Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10):771–789, 2006. (cited on pages 121 and 196).

Samenvatting

Softwareonderhoud is een van de meest kritieke activiteiten in de softwareontwikkelingscyclus (SDLC). Het bestaan van onderhoudsactiviteiten in software geeft aan dat de software nog steeds operationeel en relevant is. Onderhoud is noodzakelijk om te garanderen dat het systeem aan de veranderende verwachtingen van de gebruikers blijft voldoen. De primaire taak in softwareonderhoud is het verbeteren van fouten en het implementeren van veranderingsverzoeken.

Ongestructureerde code, onvoldoende domeinkennis en ontoereikende documentatie zijn veelvoorkomende problemen in softwareonderhoud. Deze problemen hebben een hoge impact op de prestatie van het onderhoud, omdat ze een significante invloed hebben op het begrijpen van de software. Softwarebegrip probeert kritieke informatie zoals de structuur van het systeem, het gedrag, en de interne en externe interacties van de modules uit te lichten. Onderzoek heeft aangetoond dat het begrijpen van de software de helft van de onderhoudstaak in beslag neemt. Softwaredocumentatie is een van de beste hulpmiddelen voor softwarebegrip. Echter, documentatie is vaak verouderd of in sommige gevallen niet beschikbaar.

Het achterhalen van softwarearchitectuur of softwarestructuur vanaf de implementatiecode is een gevestigd onderzoeksveld in softwareontwikkeling, genaamd Reverse Engineering. Tegenwoordig bieden diverse CASE-hulpmiddelen reverse-engineeringmogelijkheden die beloven de systeemarchitectuur terug te halen uit de broncode en deze te representeren in UML. Echter, het begrijpen van de resulterende UML-modellen is nogal ingewikkeld, omdat de gereverse-engineerde UML-diagrammen een enorme hoeveelheid informatie bevatten.

Gemotiveerd door het bovenstaande, richt dit onderzoek zich erop om een raamwerk te construeren dat het begrip van software ondersteunt door static analysis reverse engineering. Met enquêtes hebben we informatie van professionele softwareontwikkelaars verzameld over welke aspecten van een ontwerp zij indicatoren vinden voor het belang van informatie in softwareontwikkeling (wanneer deze gerepresenteerd worden met UML-klassendiagrammen).

De vondsten van het onderzoek leidden ons ertoe om objectgeoriënteerde ontwerpmetrieken en tekstmetrieken voor dit doel te gebruiken. Classificatiealgoritmes uit Machine Learning worden toegepast om een waarde te geven aan het belang van een klasse in het softwareontwerp, gebaseerd op bovenstaande metrieken. Deze machine-learningalgoritmes produceren een rangorde van klassen die de centrale informatie vormt voor het versimpelen van het gereverse-engineerde klassendiagram.

Uit de literatuur is het bekend dat verschillende belanghebbenden verschillende voorkeuren hebben voor het bestuderen van een systeem. In het bijzonder willen ze het systeem kunnen zien op meerdere niveaus van abstractie. Om dit te bewerkstelligen ondersteunt onze benadering het genereren van meerdere lagen op een schaalbare manier, dat de gebruiker in staat stelt om klassendiagrammen te bestuderen van hoog niveau van overzicht (met een klein aantal klassen) tot een gedetailleerder niveau (met een veel groter aantal klassen). Met het door ons geautomatiseerde raamwerk wordt het begrip van softwarestructuur van automatisch gereverse-engineerde klassendiagrammen praktischer. Onze oplossing draagt op deze manier bij aan het verbeteren van de efficiëntie in softwareonderhoud.

List of Publications

Below is the list of publications, which were (main) authored during this Phd research:

1. Hafeez Osman and Michel R.V. Chaudron (2011) **An Assessment of Reverse Engineering Capabilities of UML Case Tools.** *In Proceedings of the 2nd Annual International Conference on Software Engineering and Applications (SEA 2011), pages 7-12, Singapore* [Chapter 4]

2. Hafeez Osman and Michel R.V. Chaudron (2012) **Correctness and Completeness of CASE tools in Reverse Engineering Source Code into UML Model.** *GSTF Journal on Computing vol.2, num.1, pages 193-201* [Chapter 4]

3. Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2012) **Classifying Presence of Classes in UML Design using Software Metrics.** *In Proceedings of the 21st Belgian-Dutch Conference on Machine Learning (BENELEARN 2012), page 76, Ghent, Belgium*

4. Hafeez Osman, Arjan van Zadelhoff, Dave R. Stikkolorum and Michel R.V. Chaudron (2012) **UML Class Diagram Simplification: What is in the Developer's Mind?** *In Proceedings of the 2nd International Workshop on Experience and Empirical Studies in Software Modelling (EESSMod 2012), pages 31-36, Innsbruck, Austria* [Chapter 5]

5. Hafeez Osman, Arjan van Zadelhoff and Michel R.V. Chaudron (2012) **UML Class Diagram Simplification - A Survey for Improving Reverse Engineered Class Diagram Comprehension.** *In Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2013), pages 291-296, Barcelona, Spain* [Chapter 6]

6. Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2013) **An Analysis of Machine Learning Algorithms for Condensing Reverse Engineered Class Diagrams.** *In Proceedings of the 29th International Conference on Software Maintenance (ICSM 2013), Eindhoven, the Netherlands* [Chapter 7]

7. Hafeez Osman and Michel R.V. Chaudron (2013) **UML Usage in Open Source Software Development : A Field Study.** *In Proceedings of the 3rd International Workshop on Experience and Empirical Studies in Software Modelling (EESSMod 2013), pages 23-32, Miami, USA* [Chapter 3]

8. Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2014) **Interactive Scalable Abstraction of Reverse Engineered UML Class Diagrams.** *In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014), Jeju, Korea* [Chapter 9 and 10]

9. Hafeez Osman, Michel R.V. Chaudron and Peter van der Putten (2014) **Condensing Reverse Engineered Class Diagrams through Class Name Based Abstraction.** *In Proceedings of the 2014 World Congress on Information and Communication Technologies (WICT), Malacca, Malaysia* [Chapter 8]

The following is the publication which were co-authored and are related to this thesis. However, these works are not included (in this thesis) due to secondary contribution.

10. Ferdian Thung, David Lo, Hafeez Osman and Michel R. V. Chaudron (2014) **Condensing Class Diagrams by Analyzing Design and Network Metrics using Optimistic Classification.** *In Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014), Hyderabad, India*

11. Truong Ho Quang, Michel R.V. Chaudron, Ingimar Samúelsson, Jóel Hjaltason, Bilal Karasneh and Hafeez Osman (2014) **Automatic classification of UML Class diagrams from images.** *In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014), Jeju, Korea*

Acknowledgments

Over the past four years, many people and organizations have contributed to the completion of this thesis. First and foremost, I want to thank my sponsor, the government of Malaysia, for giving me the opportunity to enhance my knowledge and experience through this Ph.D journey. I wish the knowledge that I have gained through this journey would benefit our lovely country.

To my colleagues (and ex-colleagues) in Software Engineering Section, LIACS: Ana Fernandez, Ariadi Nugroho, Bilal Karasneh, Dave Stikkolorum, Ramin Etemaadi and Werner Heijstek. Thank you for your kind advice and non-stop support to this research (also regarding living in the Netherlands). I feel really fortunate to be part of this group.

My roommates in LIACS: Ali Mirsoleimani and Ben Ruijl. Thanks for the support and advice. I enjoyed our 'lunch meetings' and discussions.

LIACS students: Arjan van Zadelhoff, Wei Liu and Mahya Mirtar. Thank you for helping me with the research and the tool. Other LIACS (ex-)members: I would like to thank Mohamed Bamakhrama, Mohamed Tlais, Abdel, ShengFa, Ricardo, Rafael, Alex, Di Liu, Jelena, Emanuel, Asep Maulana, Frank and Marijn. Also, I would like to thank other LIACS members that participate in the survey and experiments in this research. Thank you for your time and efforts.

To my friends in Chalmers: Abdullah, Truong, Antonio, Imed, Håkan and Grisha, thank you for helping me with the research experiments, research collaborations and your warm welcome during my visits there.

I also want to take this opportunity to express my gratitude to our collaborator, Dr. David Lo and Ferdian Thung from Singapore Management University. Thank you for your ideas on how to improve the method invented in this research.

To (ex-) Malaysian In Leiden: Thank you Zuwairi and Farah, Liew and Wong, Hidayah, Che Zuhaida, Fadzrul and Zarina, Dennis and Kak Emma, Casper and Kak Jie, Dr.

Azman and Dr. Nik, Norlela, Naqiuddin, Yuvendran, Shafa'atussara, Syibli and Alia, Roslina and Zaleha. I am glad to be part of our Malaysia family here.

To my family: My parents, Khelah Mohd Idris, Osman Hassan, and Zaiton Sharif, also my parents-in-law, Jamaludin Bador and Aznor Yacob, thank you for your prayers, love and support. I would like to thank my sisters Hafeza and Insyirah, and my brother Fakhri for their prayers and for taking a good care of our lovely mother while I am not around.

The biggest thank you is to my lovely wife, Normal Aznita. I really appreciate your patience, understanding and sacrifice. Thank you for completing my life. Last but not least, I would like to thank my kids, Ash-Syifa and Firas for always throw me their lovely smile that always makes me happy.

Mohd Hafeez Osman
Leiden, March 2015

About the Author

Mohd Hafeez Osman was born on April 20, 1979 in Ipoh, Malaysia. He graduated his B.Sc. (hons) in Computer Science (major: computer system) from the University Teknologi Malaysia (UTM) in 2001. Then, he immediately continues his study and graduated his M.Sc. in Computer Science – Real Time Software Engineering in 2003 from the same university.

His first employment was in August 2002 as a software engineer for a software company. After working for several companies, in 2004, he started working as the Information Technology (I.T.) Officer for the government of Malaysia. After being working for eight years in the I.T. industry, he received a Malaysian Government Federal Training Award (HLP). This award allows him to take a four years study leave with a full Ph.D scholarship. On September 2010, he started his Ph.D journey at Leiden Institute of Advanced Computer Science (LIACS), under the supervision of Prof. Dr. Michel R.V. Chaudron and Dr. Peter van der Putten.

After finishing his Ph.D study, Hafeez will resume his job as the I.T. officer for the Malaysia government and continue his interest in Software Engineering, Artificial Intelligence and Computer Networks.