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

Osman, M.H.B.

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.

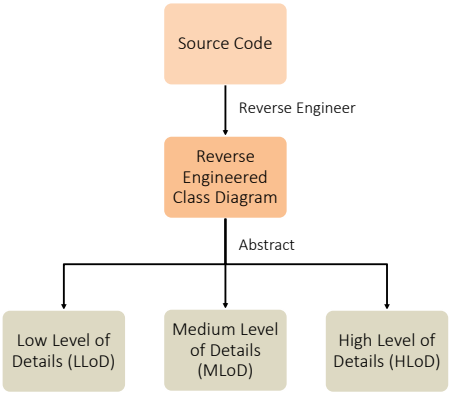


Figure 5.1: *Level of Detail Class Diagrams Preparation*

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.

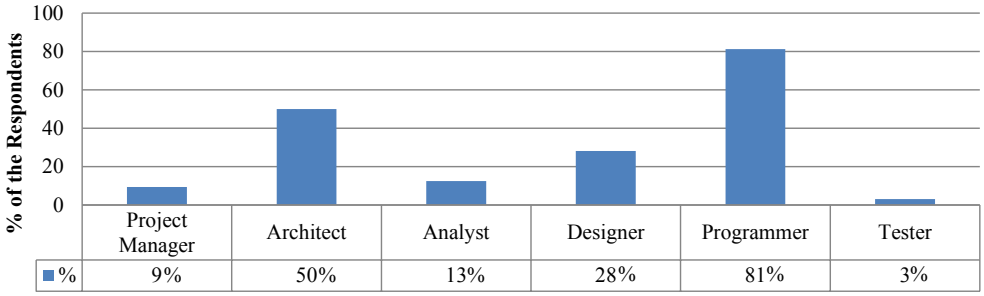


Figure 5.2: *Role of the Respondents*

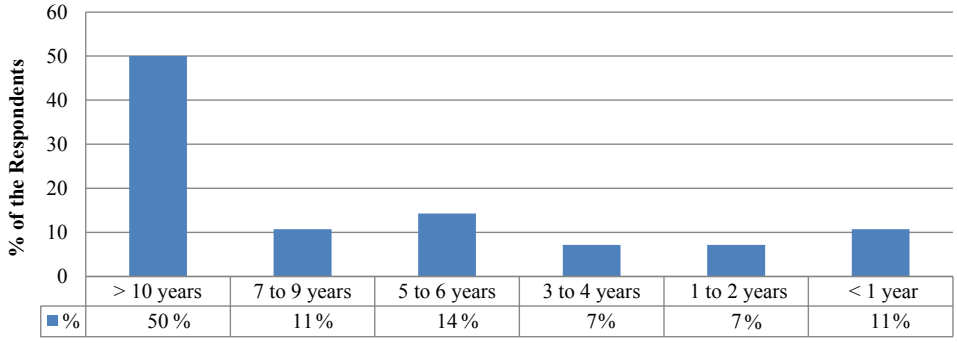


Figure 5.3: *Respondents Experience with Class Diagrams*

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

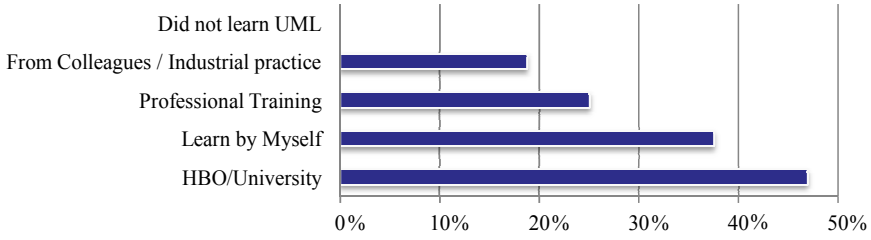


Figure 5.4: *Where did the Respondent Learn about UML*

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

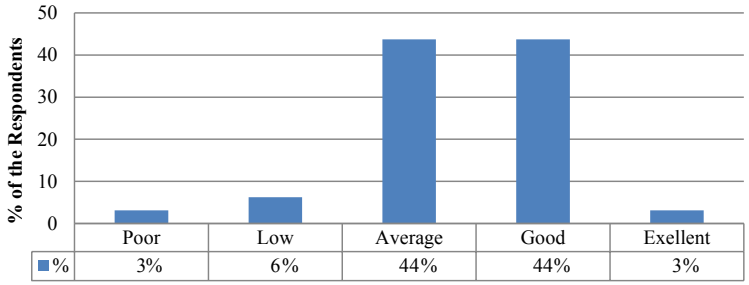


Figure 5.5: *Respondent's skill on Class Diagram*

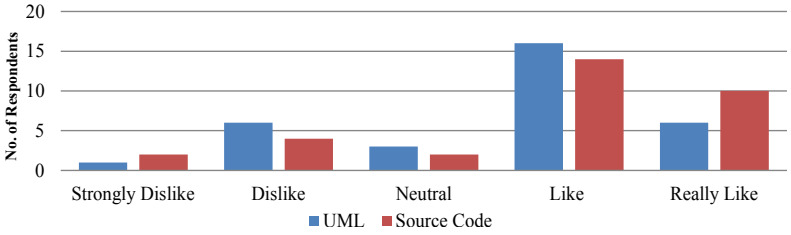


Figure 5.6: *Respondents Like or Dislike Source Code vs UML*

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

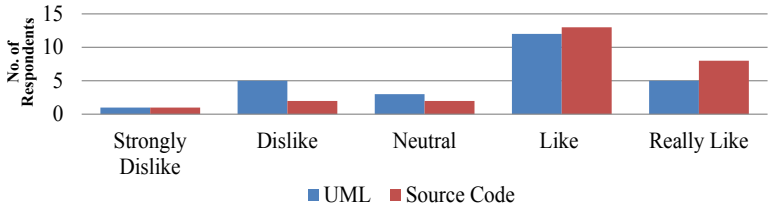


Figure 5.7: *Programmers Like or Dislike Source Code vs UML*

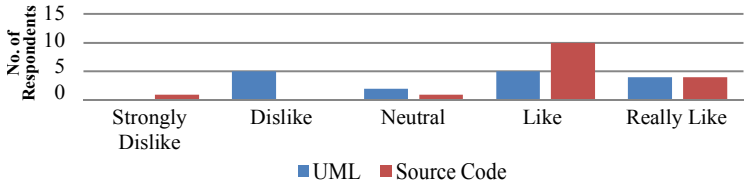


Figure 5.8: *Software Architects Like or Dislike Source Code vs UML*

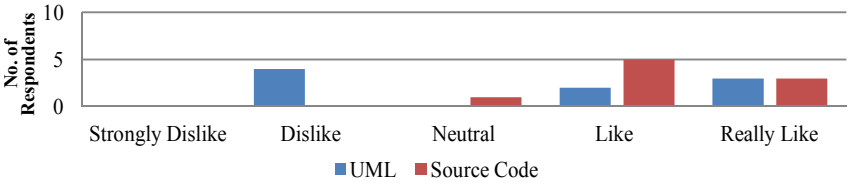
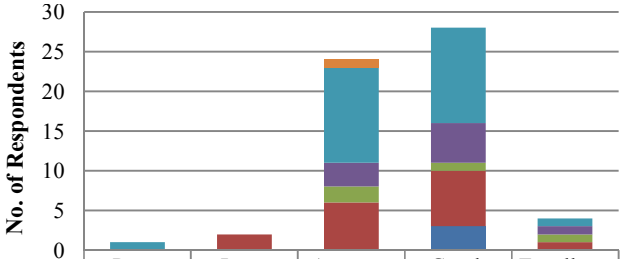


Figure 5.9: *Software Designers Like or Dislike Source Code vs UML*



	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

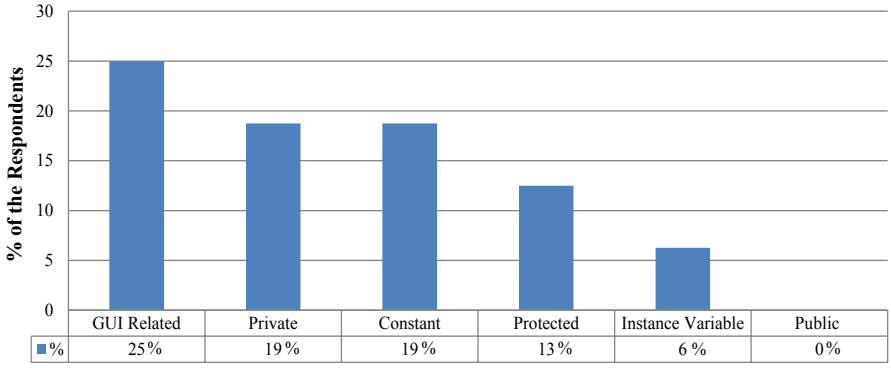


Figure 5.11: *Information of Attribute that Could be Left out*

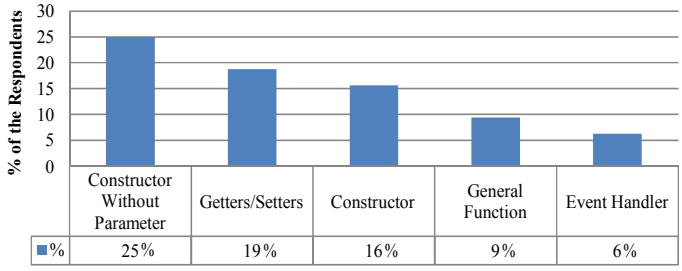


Figure 5.12: *Types of Operation that could be Excluded in Class Diagrams*

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.

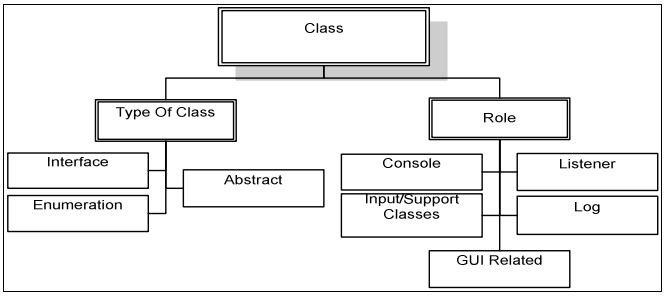


Figure 5.13: *Class Category*

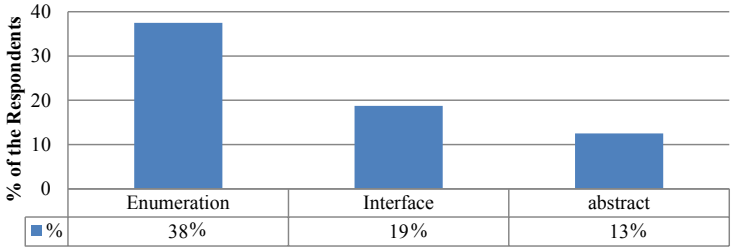


Figure 5.14: *Types of Class that could not be Included in Class Diagrams*

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.

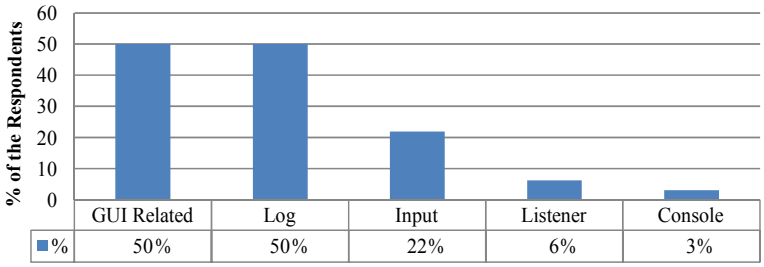


Figure 5.15: *Class Role that could be Excluded in Class Diagrams*

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.

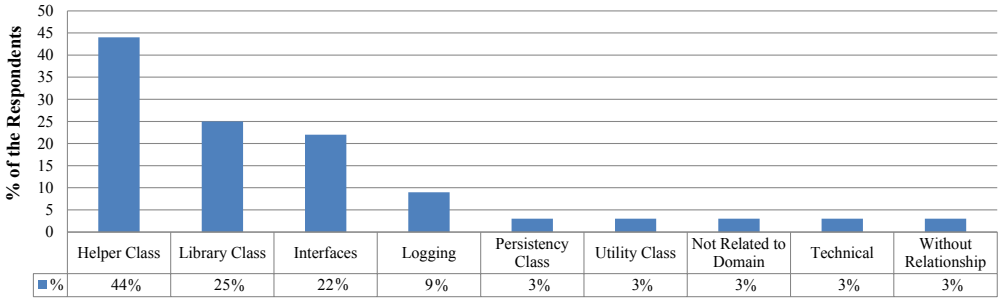


Figure 5.17: *Information of Classes that could be Omitted*

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

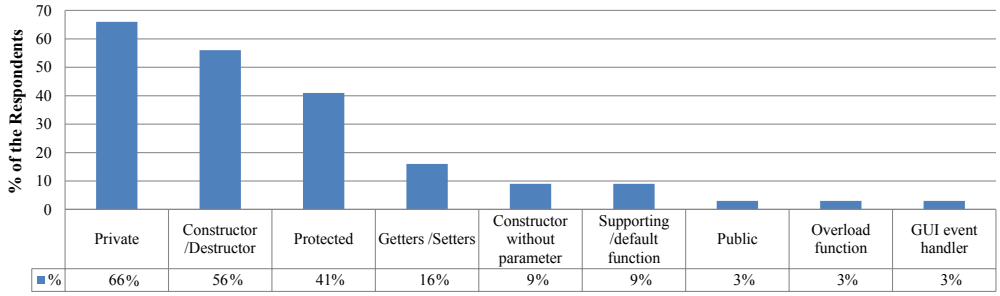


Figure 5.18: *Information of Operations that could be Omitted*

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.

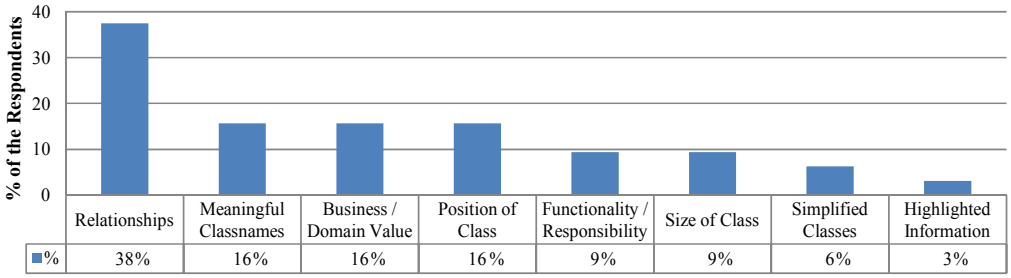


Figure 5.19: *Important Criteria in a Class Diagram for Understanding a System*

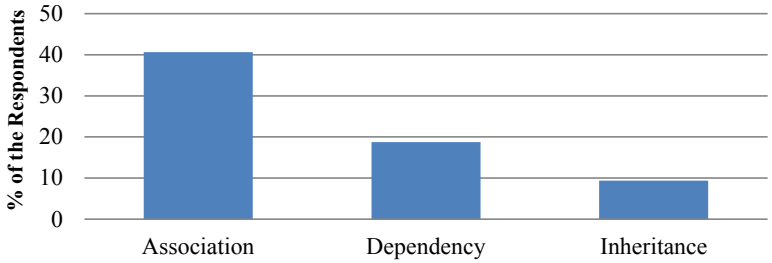


Figure 5.20: *The Type of Relationship in Class Diagrams that the Respondents Look at First*

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.

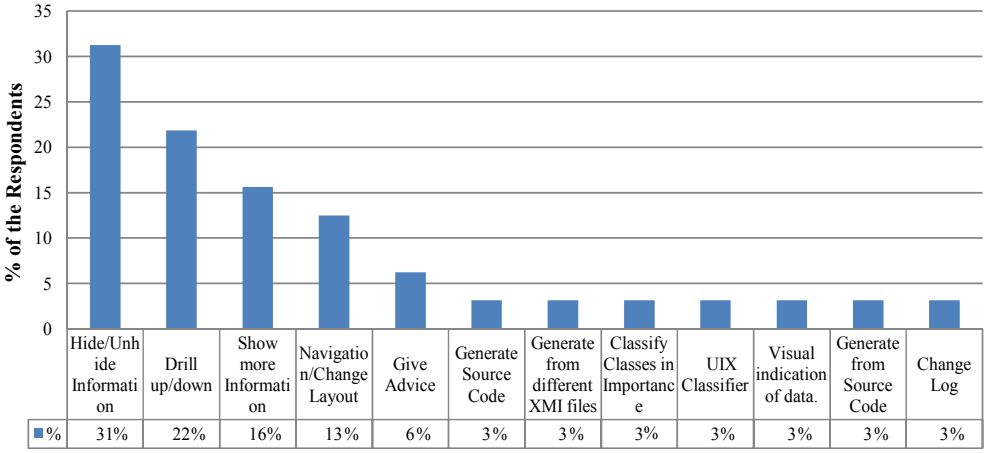


Figure 5.21: *The Features that a Tool Should have for Simplifying UML Class Diagrams*

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.