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

Osman, M.H.B.

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.