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

Osman, M.H.B.

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.

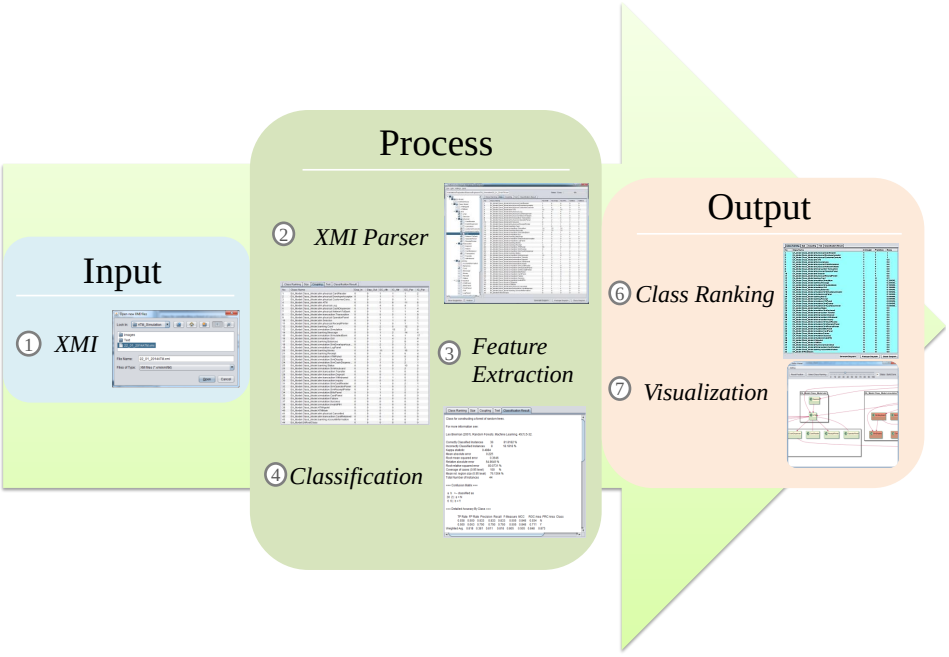


Figure 9.1: *Overall Framework : Input, Process and Output*

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.

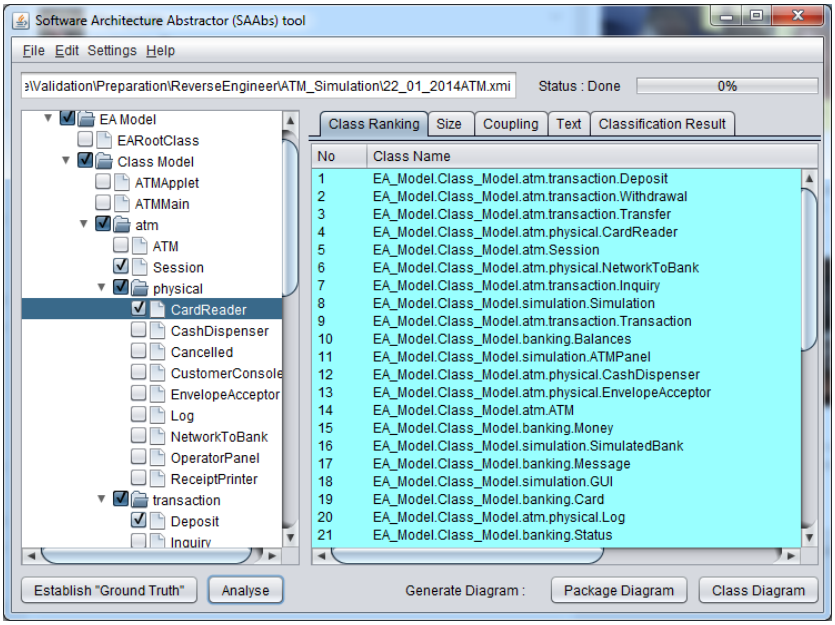


Figure 9.2: *Selection of the Candidate-Important Classes*

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.

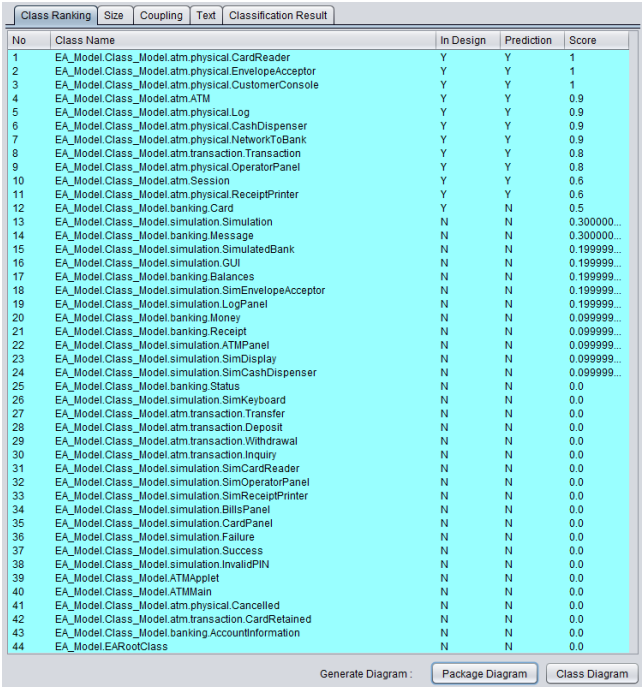
```
1: Input :
2: Document (D) = Design Document (XMI)
3: Output :
4: ListImpClsCand = Table of Class Metrics
5: Method :
6: Dep_In = Dependency In
7: Dep_Out = Dependency Out
8: EC_Par = Export Coupling Parameter
9: EC_Attr = Export Coupling Attribute
10: IC_Par = Import Coupling Parameter
11: IC_Attr = 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.

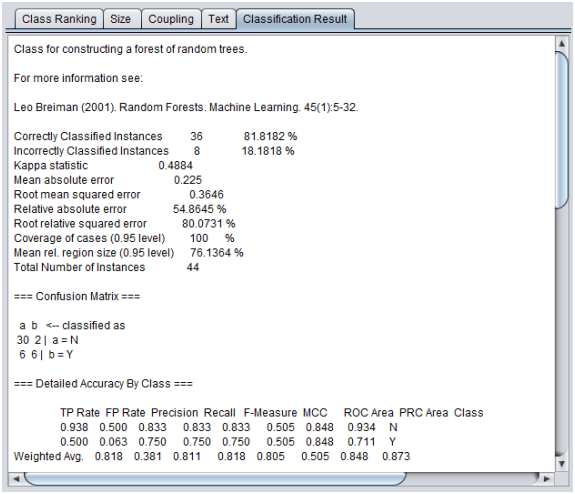


Figure 9.4: *Textual Results of Classification*

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.

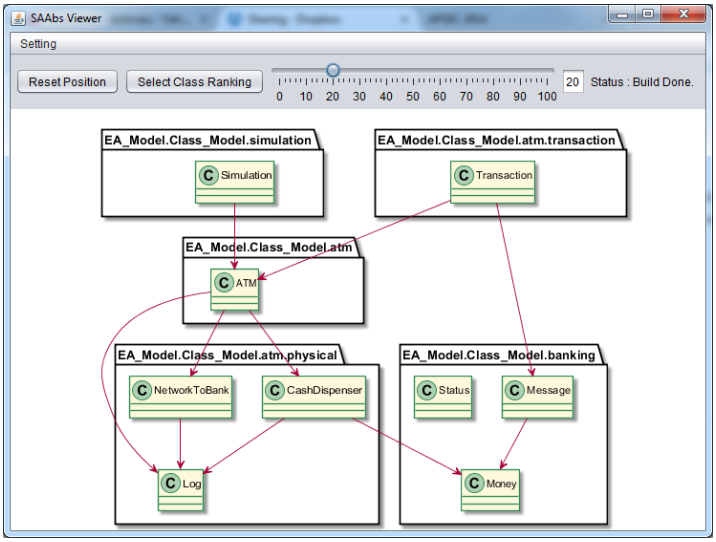
2. *Hiding Class Diagram Elements*: In order to simplify the class diagram, the respondents in [133] have mentioned that some class diagram elements may be hidden: a) getters and setters, and b) constructor. Thus, SAAbs offers the option to hide the getters and setters, constructors, and also attributes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

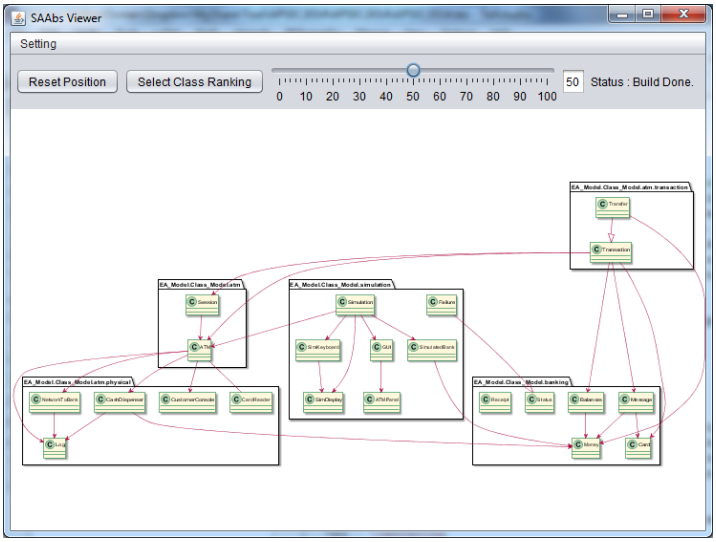
9.3.7 Implementation

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*

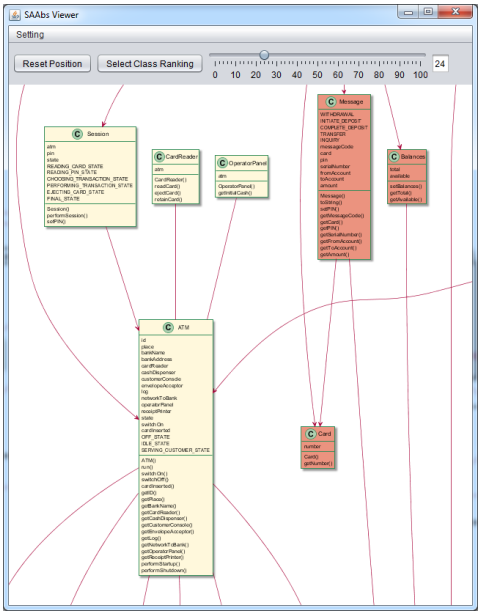


Figure 9.6: *Highlighting of Less Important Classes*

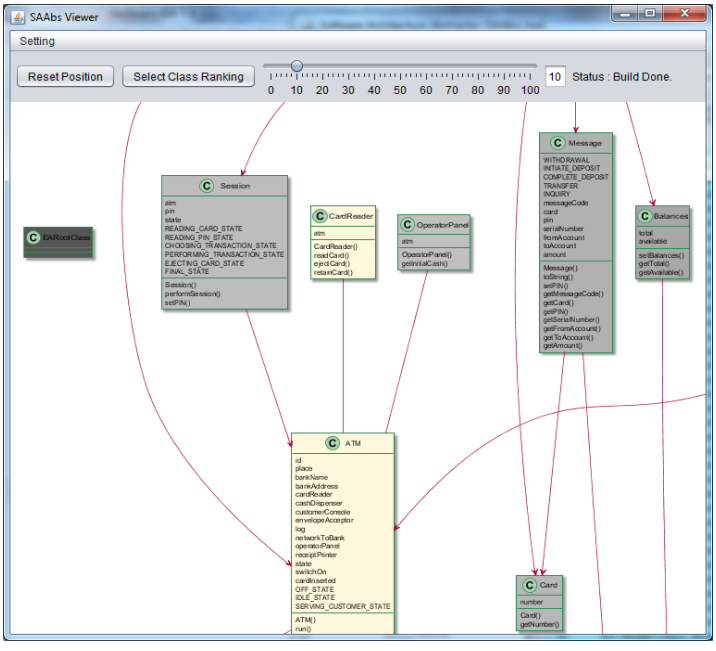


Figure 9.7: *Greyscale Coloring: Less Important Classes with Darker Shades of Gray.*

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).