EXECUTION TRACE ANALYSIS USING UTILITY CLASS DETECTION AND DECOUPLING IN OBJECT-ORIENTED SOFTWARE COMPREHENSION

HASAN MUGBIL KHALAF ABU AL ESE

UNIVERSITI SAINS MALAYSIA

2015

EXECUTION TRACE ANALYSIS USING
UTILITY CLASS DETECTION AND DECOUPLING
IN OBJECT-ORIENTED SOFTWARE COMPREHENSION

by

HASAN MUGBIL KHALAF ABU AL ESE

Thesis submitted in fulfillment of the requirements
for the degree of
Doctor of Philosophy

July 2015

ACKNOWLEDGMENTS

The praises are due to ALLAH, the most merciful, the most compassionate, and may the prayers of blessing of Allah be upon Prophet Muhammad, the chosen, the trustworthy, and upon his family and all of his companions.

I would like to express my deepest grateful thanks and gratitude towards my thesis advisor Associate Professor Dr. Putra Sumari, for his extraordinary supervision, friendship, invaluable guidance and constructive criticism which made this work possible. He pushed me when I needed to be pushed. I am thankful for every minute he spent on giving me guidance to achieve my academic goal. He has taken so much of his valuable time for reading, correcting and restructuring the preliminary drafts. I would also like to thank the members of my examination committee for the feedback they provided during my PhD viva voce. Iam also indebted to Professor Dr. Ahamad Tajudin Khader and Associate Professor Dr. Shahida Binti Sulaiman.

I would like to convey my appreciation to the School of Computer Sciences in University Sains Malaysia (USM), the Library of the University, the Institute of Postgraduate Studies (IPS) and the Laboratories Technicians. I am also grateful to my colleagues for their encouragement with this work.

Last but not least, I would like to express my most sincere and warmest gratitude to my mother, wife, children, sisters, brothers, sisters in law, brothers in law, uncles, aunts, cousins, nephews, and nieces for their prayers, love, generous moral and financial support during my study.

Thank You All.

TABLE OF CONTENTS

PAGE

LIST OF TABLES

PAGE

LIST OF FIGURES

PAGE

LIST OF ABBREVIATIONS

C	Correctness of solutions
JIDE	Java IDE
MT	Maintenance Tasks
PC	Program Comprehension
SE	Software Engineers
SS	Software System
T	Time spent
UD	Utility Decoupling Tool
US	Usability and Usefulness Survey

ANALISA SURIH PELAKSANAAN MENGGUNAKAN PENGESANAN DAN PENYAHGANDINGAN KELAS UTILITI DALAM PEMAHAMAN PERISIAN BERORIENTASIKAN OBJEK

ABSTRAK

Sistem perisian berorientasikan objek adalah platform yang paling banyak digunakan dalam organisasi di dunia pada hari ini. Penyelenggaraan sistem ini sememangnya menjadi satu tugas yang penting untuk memastikan sesuatu perisian sentiasa dikemaskini dan selari dengan perubahan pada beban kerja dan pembaharuan teknologi. Salah satu kaedah untuk melakukan penyelenggaraan ini adalah untuk menyurih pelaksanaan sistem dan kemudian menganalisanya yang dipanggil sebagai teknik analisa surih pelaksanaan. Walau bagaimanapun, sistem perisian berorientasikan objek mempunyai pelbagai kelas dan ciri gandingan yang membuat analisa menjadi sukar. Surih pelaksanaan sistem perisian pada masa kini cenderung untuk menjadi sangat besar dari segi kerumitan dan saiz. Kebergantungan antara kelas-kelas dan ciri-ciri gandingan membentuk jalinan kekisi yang sangat rumit. Ini berkaitan terutamanya dengan utiliti yang sememangnya lebih boleh diguna semula dan mempunyai penyahgandingan yang sangat kukuh. Tesis ini memperkenalkan satu teknik analisa surih baru yang ringkas dan memudahkan proses menyurih pelaksanaan. Kerja yang dicadangkan terdiri daripada tiga komponen utama, iaitu komponen penapisan skop, komponen pengesanan kelas utiliti, dan komponen penyahgandingan kelas utiliti. Komponen penapisan skop adalah untuk menapis modul aplikasi yang tidak dikehendaki bagi memilih hanya skop-skop tertentu dalam sesuatu sistem perisian untuk aktiviti-aktiviti surih pelaksanaan. Komponen pengesanan kelas-kelas utiliti bertujuan untuk mengesan kelas utiliti dalam skop

tertentu yang telah dipilih untuk aktiviti-aktiviti penyurihan. Di sini, dua metrik pengesanan kelas utiliti baru dicadangkan. Metrik-metrik ini bergantung terutamanya kepada gandingan dinamik untuk merekod ciri-ciri masa laksana sistem berorientasikan objek seperti polimorfisma dan pengikatan lewat. Akhir sekali, komponen penyahgandingan kelas utiliti memisahkan gandingan antara kelas-kelas utiliti. Kerja yang dicadangkan dinilai secara kuantitatif dengan menggunakan eksperimen terkawal. Eksperimen menunjukkan peningkatan 25% dalam pengurangan masa yang digunakan dan 62% ketepatan penyelesaian untuk menjawab tugasan kefahaman yang diberikan. Selain itu, perbandingan dengan kaedah yang lain dalam bidang yang sama telah dijalankan. Perbandingan menunjukkan peningkatan 15% masa yang dikurangkan dan 12% ketepatan penyelesaian untuk menjawab tugasan kefahaman yang diberikan. Keputusan mengesahkan kecekapan dan keberkesanan kerja yang dicadangkan untuk membuat surih pelaksanaan kurang sukar.

EXECUTION TRACE ANALYSIS USING

UTILITY CLASS DETECTION AND DECOUPLING

IN OBJECT-ORIENTED SOFTWARE COMPREHENSION

ABSTRACT

Object-oriented software systems are the most used platforms in most today organizations in the world. The maintenance of these systems indeed is becoming an important task in order to assure the software keep updated with changes of the recent workload and technologies. One method to do the maintenance is to trace the executions of the system and yet analyze them which is called execution trace analysis technique. However, object-oriented software has classes and coupling features that make the analysis difficult. The execution traces of current software systems tend to be very large in terms of complexity and size. The classes and coupling features form a very complicated interwoven lattice of the dependencies. This applies particularly to utilities which are inherently more reusable and having very tight coupling. This thesis introduces a new trace analysis technique that simplifies and eases the execution tracing process. The proposed work consists of three main components, namely scope filtering component, utility class detection component, and utility class decoupling component. The scope filtering component filters the unwanted application modules to yield only a specific scope of the software system for the execution trace activities. The utility class detection component detects the utility classes within a particular scope of a given execution trace. Here, two new utility class detection metrics are proposed. These metrics depend mainly on the dynamic coupling to capture runtime properties of an object-

oriented system such as polymorphism and late binding. Lastly, the utility class decoupling component decouples the tightly coupled utility classes. The proposed work is evaluated quantitatively using a controlled experiment. The experiment showed an improvement of 25% less time spent and 62% correctness of solutions to answer given comprehension tasks. Moreover, comparisons with related state-of-art methods are conducted. The comparisons showed an improvement of 15% less time spent and 12% correctness of solutions to answer given comprehension tasks. The results verify the efficiency and effectiveness of the proposed work in order to make execution traces less difficult.

CHAPTER 1

INTRODUCTION

1.1 Program Comprehension

In the area of software engineering, program comprehension is an extremely essential activity of software maintenance to get better understanding of software systems before they can be modified (Demeyer et al., 2003; Ko et al., 2006; Sommerville, 2011).However, program comprehension has applications in other software engineering areas such as software development, software reuse, software migration and software reengineering (Obrien, 2003; Storey, 2006). The area of program comprehension is also known as software understanding. Therefore, the terms, comprehension and understanding are used as synonyms.

Actually, program comprehension process is an extremely individual process. For example, several software engineers may use the same way in understanding the software systems, nevertheless, the results may vary from one software engineer to another. Therefore, several definitions are found in the literature to identify what program comprehension means. Among these definitions, one could recognize the definition introduced by Zhang (2005) as follows:

> *"Program comprehension is the process of deriving from program code abstract information which are meaningful to engineers and can help them to learn the program, making decisions and modify the program correctly."*

1.2 Coupling and Complexity

Coupling is a powerful technique for assessing relationships among software entities to understand how they are relate to each other before any modification. In coupling, two entities are coupled when they are related to each other by any kind of relationship or connection (Abdurazik, 2007). Coupling as a metric, was first introduced by Stevens et. al. (1974) as the measure of connection strength that is established between two modules. The concept of coupling has been adapted to object-oriented software by Coad and Yourdon (1991) and numerous metrics for object-oriented software have been defined.

However, coupling is related directly to complexity (structural complexity rather than computational complexity) in a positive correlation. For example, tight coupling leads to high complexity as components are more inter-related whilst, loose-fitting coupling leads to low complexity where the components are less inter-related. In particular, current object-oriented systems lead to form a very complicated interwoven lattice of dependencies which is known as "Spaghetti Architectures" phenomenon (Webster and Simon, 2011). The reason is that when high performance and fast turnaround time are crucial to a system, components are intentionally programmed to be tightly coupled. In this case, the functions in each of the tightly coupled components are cohesive. However, a very tight coupling implies a complicated structure of the system, therefore, the structural complexity is expected to be very high.

Complexity refers to the degree of difficulty to understand and verify a software system or one of its components (IEEE glossary, 1990). In the literature,

complexity is represented by several properties such as size and coupling. However, the size property cannot sufficiently characterize the structural complexity as any two different software systems of similar size are almost different in structure. Alternatively, coupling is a good indicator for the structural complexity as coupling can depict the hierarchy of the system and the structural dependencies between its components.

Similar to complexity, coupling has a negative impact on program comprehension and software maintenance. The reasoning is that, when a component is coupled to more other components, this means that in order to understand that particular component, more "links" and components need to be investigated which makes understanding it more difficult. Also, assembly of coupled components might require more effort and/or time due to the increased inter-components dependency. The reasoning is also similar for maintenance, when a maintainer wants to change a component, but is coupled to many other components, the ripple effect might be bigger and/or additional constraints for making changes might apply. Hence, it is desirable to keep coupling as loose as possible in order to ensure that changes to one component have limited impact on the rest of other components. However, coupling is unavoidable within a software system as components need to work together to achieve the desired functionality.

1.3 Background of the Problem

Current software systems tend to be very large in terms of complexity and size. Consequently, maintenance of these software systems requires exploiting various knowledge resources such as the availability of the original developers and

up-to-date documentation. Otherwise, the maintenance process will be tedious, costly and time-consuming. However, original developers usually switch to a new system or even a new firm after the current system has been delivered and up-to-date documentation is often not available or insufficient. These documentation problems may be attributed to time-to-market constraints, excessive ad-hoc maintenance activities and the cost of updating is not justifying the benefits. Thus, Software maintenance activities consume about two-thirds of the budgets of IT systems, which are considered to be a very high cost proportion (Sommerville, 2011).

The major factor that leads to this higher cost proportion is the understanding process of the software system under maintenance. In particular, more than half of the maintenance costs are assigned to understand the intended software system (IEEE CS, 2012). For example, a considerable amount of time, required for maintenance process, is spent in understanding the software system and analyzing the impact of the proposed changes (Storey, 2005). Therefore, understanding of an existing software system is a costly activity, in particular, when software systems undergo several maintenance cycles (Hamou-Lhadjand Lethbridge, 2010). Thus, there is a need to develop tools and techniques that support the comprehension process. These tools and techniques should rely on reliable and complete up-to-date references which may be confined only to programcodes (i.e., source code and object code).

In the literature, there are two main comprehension techniques for program analysis namely, static analysis and dynamic analysis. The static analysis techniques are based on parsing the source code of a program without executing it. Whilst, dynamic analysis techniques are based on analyzing the dynamic behavior of the

program by extracting its dynamic information while it is executed. Hence, the static techniques analyze what may possibly occur (i.e. they examine all execution paths) whereas the dynamic techniques analyze what is actually occurring (i.e. they examine only actual execution paths for a particular execution scenariothat contains one or more features). Therefore, both of the dynamic and static techniques are complementary (i.e. one of them cannot supplant the other) and they have the ability to make the understanding process easier and less costly.

However, the key issues of object-orientation such as polymorphism and late binding necessitate the use of dynamic analyses where the actual polymorphic method calls can only be determined at runtime (Zaidman and Demeyer, 2008; Chhabraand Gupta, 2010; Gupta, 2011).In addition, dynamic analysis techniques can support goal-oriented comprehension strategy that allows maintainers focus only on interested parts rather than taking the entire system into consideration. Hence, the dynamic analysis techniques have the potential of providing precise structure of software systems through addressing runtime information that is commonly represented in the form of execution traces. Figure 1.1 shows an example of polymorphism in a Java program where there are several methods that have an identical name, that is, open(). Therefore, the executed behavior is determined at the runtime, not at the compile time. The only way to determine which of a number of dynamic binding actually occurs in a particular set of circumstances is to trace through the code, either by running it on a computer or tracing through it manually. However, manually tracing for a large set of objects with intensive use of polymorphism and late binding is difficult task, if possible at all.

```
class Driver {
      static public void main (String[] args) {
       File myFile;
       .
       .
       .
       .
      myFile.open();
      }
}
abstract class File {
      abstract void open();
}
class DiskFile extends File {
      void open() {
           System.out.println("open file from a disk");
      }
}
class TapeFile extends File {
      void open() {
           System.out.println("open file from a tape");
      }
}
class DisketteFile extends File {
      void open() {
           System.out.println("open file from a diskette");
      }
}
```

Figure 1.1: Example of Polymorphism.

1.4 Research Problem

Coping with execution traces is a daunting task as they are tend to be very large in terms of size and complexity (Cornelissen, 2009; Hamou-Lhadjand Lethbridge, 2010; Pirzadeh, 2012). For example, an execution trace for a current industrial system, usually consists of several thousands up to several millions of events (Dugerdil and Repond, 2010). In addition, current object-oriented systems lead to forma very complicated interwoven lattice of dependencies which is known as "Spaghetti Architectures" phenomenon (Webster and Simon, 2011). Therefore, the major challenge for the trace analysis techniques is how to properly convey the large and complicated traces to the maintainers.

Trace analysis techniques involve trace visualization and trace reduction in order to simplify the understanding of large and complicated traces and yet minimize

effort and time needed for the maintenance process. Unfortunately, trace visualization techniques are being limited in several cases. In particular, they require considerable intervention from the user side to analyze the intended trace. In other words, it is absolutely up to the user to navigate and explore among the diversity of features in the trace (Pirzadeh, 2012). Also, these techniques in most cases have upper limits on the amount of data that can be tackled (Cornelissen, 2009). Consequently, the problem of analyzing large and complicated traces is turned to as the problem of developing visualization tools. These limitations necessitate the use of trace reduction in order to: 1) make execution traces more tractable and less difficult; 2) protect users from being confused by massive data of traces; and 3) alleviate user intervention and interpretation. However, trace reduction techniques should be adequately introduced in order to produce informative traces, otherwise, the reduction process be useless.

Unfortunately, most trace reduction techniques represent the problem by trace sizeonly without paying attention to its structural complexity. However, the size property cannot sufficiently characterize the trace. For example, any two different traces of similar size are almost different in structure. Also, trace components are usually not equally important. In particular, there exist components that complicate the relationships between various trace componentswith little or no important to program comprehension. Consequently, they hinder the understanding process (Pirzadeh et al., 2009; Hamou-Lhadjand Lethbridge, 2010).This applies particularly to utilities which are inherently more reusable components. Hence, they have very tight coupling and yet raise the structural complexityof execution traces.

1.5 Objectives of the Research

The main objective of this thesis is to propose a new trace analysis method that can effectively simplify understanding of large execution traces. The specific objectives of this thesis include the following:

1. To establish a new trace simplification framework, that can reduce the impact of trace complexity in order to facilitate the understanding process for maintenance tasks and reduce the relevant costs and time.

2. To detect utility componentsprecisely using dynamic couplingin order to determine which components lead to higher complexity.

3. To decouple utility components using module facade in order to resolve the trace complexity problem without creating gaps and holes in components dependencies.

1.6 Scope of the Research

The scope of this research is bounded as follows:

1. The domain of this research is software engineering.

2. The application of the proposed work is scoped to the understanding of execution traces to perform some maintenance tasks. In particular, this thesis focuses on adaptive maintenance, where maintenance tasks may involve the modification of existing features or the addition of new ones.

3. The trace mode used in this research is restricted to offline mode.

4. The trace type used in this research is limited to "method calls" in Java systems.

1.7 Research Contributions

The major contributions of this thesis are as follows:

1. A trace simplification framework is proposed based on a combination of three components.The three components are scope filtering, utility class detection, and utility class decoupling. The proposed framework provides further manipulation to the execution trace and it can provide different views of the software system at different runs.

2. Two new metrics for utility class detection based on dynamic coupling are proposed. The first metric considers only the coupling in one direction (i.e. export coupling). Whilst, the second one considers both directions of coupling (i.e. export and import coupling). The importance of the proposed metrics is that they can detect utility classes precisely in object-oriented systems as intensive use of polymorphism and late binding. Moreover, they can detect utility classes at distinct scope of software systems such as feature scopes.

3. A scheme for utility class decoupling based on a modified module façade is proposed. The proposed scheme comprises five main components namely trace collection component, class identification component, filtering component, decoupling component and finally trace simplification component. This scheme utilizesthe proposed utility class detection metrics to find utility classes in a particular execution trace, and then facadeprocessing is applied to perform the decoupling. It composes a subsystem facade consisting of the detected utility classes.Then, itencapsulates and hides that subsystem behind a coordinator interface.

1.8 Organization of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 discusses the different topics that are related to this research as well as provides a review of the existing trace analysis techniques for program comprehension. The chapter starts with presenting software maintenance. Also, it presents program comprehension and its cognitive models. Then, dynamic analysis and its importance are provided. The chapter continues with mapping between coupling and program comprehension. In the meantime, it provides a comparison between static and dynamic coupling as well as the classification of dynamic analysis. Then, the different decoupling strategies are briefly presented with the focus on modularity patterns as means of performing decoupling. Afterward, the chapter proceeds with reviewing in details the trace analysis tools and techniques available in the literature. In particular, six trace analysis tools are reviewed in order to infer the embedded techniques. These tools include content prioritization, EXTRAVIS, trace summarization, shimba, AVID and together diagrams. The reviewing process involves clarifying the strengths and weaknesses of each tool. Finally, the inferred trace analysis techniques are discussed and their pros and cons are provided.

Chapter 3 provides an overview of the research methodology of this thesis. The chapter starts with identifying the research procedure. Then, it presents the research justification. Also, the evaluation method is introduced where a controlled experiment is conducted to quantitatively measure the efficiency and the effectiveness of the proposed work. In addition, comparisons with related state-of-art methods are conducted. Finally, the limitations and lists of assumptions are provided.

Chapter 4presents the proposed trace simplification framework and its major components. The chapter starts with identifying the theoretical framework. Then, it introduces the proposed trace simplification framework.After that, it illustrates the major components of the proposed framework and how walking through them. Finally, it presents the prototype tool which implements the proposed work.

Chapter 5 presents the proposed utility class detection metrics and how to exploit dynamic coupling for this purpose. The chapter starts by mapping between utility detection and dynamic coupling. Then, it presents the proposed utility detection metrics. In particular, two new utility class detection metrics are proposed.

Chapter 6presents the proposed utility class decoupling scheme and how to utilize modularity patterns in order to perform the decoupling. In particular, module facade is utilized. The chapter starts by providing a discussion on the overview of the proposed scheme then, elaborates the details. Also, it illustrates the main steps of the algorithm and the setting of the required parameters.Finally, it describes the UtilityDecoupling tool in more detail.

Chapter 7 validates this research empirically by conducting a controlled experiment. It also provides the results and discussion of this controlled experiment as well as its usefulness and usability. In addition, the chapter provides a comparison between the results of the controlled experiments in this thesis and the results of Extravis trace visualization experiment in order to infer "lessons learned" about trace simplification and trace visualization.

Finally, Chapter 8 concludes and highlights the major contributions of this thesis and presents plans for future possible work.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Dynamic analysis techniques are used to extract and analyze systems behavior to facilitate program comprehension. This research is intended to improve the efficiency and effectiveness of such techniques by helping software maintainers to understand the content of large execution traces. This Chapter consists of two main parts. Section 2.2 through section 2.7 present related background topics that are necessary to understand this thesis. These topics include software maintenance, program comprehension, dynamic analysis, coupling, utilities and decoupling. Section 2.8providesa review of state-of-art trace analysis tools,and then discusses their strengths and weakness.The inferred trace analysis techniques are discussed in section 2.9.Finally, section 2.10 summarizes this chapter.

2.2 Software Maintenance

Software maintenance is a central part of software evolution and it is an inevitable process to remain software systems useful (Sommerville, 2011). Software systems must be continually adapted otherwise they become progressively less satisfactory in use. Consequently, software engineering is a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the software systems. For example, once the first release of the software system is

delivered, enhancements are proposed and the development of the second release starts shortly (Sommerville, 2011).

IEEE CS (2012) defines software maintenance as the totality of activities required to provide cost-effective support to software. Activities involve the ones that carried out during both the pre-delivery and post-delivery stages. Pre-delivery activities include maintainability, planning for post-delivery operation and logistics determination for transition activities whereas post-delivery activities include training, modification and operating or interfacing to a help desk. Moreover, Pfleeger and Atlee (2009) state that maintenance has a broader scope, with more to track and control than development. For example, software maintenance includes understanding the existing systems, documenting systems, extending existing functions, adding new functions, finding and correcting faults and bugs, helping and training users, restructuring and purging software systems, managing the software systems, and all other activities that go into running successful software systems. Thus, Software maintenance activities consume about two-thirds of the budgets of IT systems, which is considered to be a very high cost proportion (Sommerville, 2011).

IEEE Standards 14764 (2006) definesfour categories of maintenance as follows:

1. Corrective maintenance: reactive modification of a software product performed after delivery to fix faults that cause the software to fail.
2. Adaptive maintenance: modification of a software product performed after delivery to keep software product usable in a change or changing environment.

3. Perfective maintenance: modification of a software product after delivery to improve its performance and to improve its flexibility to make it easier to extend and add new features in the future.
4. Preventive maintenance: modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

Actually, these categories have no explicit distinction between each other. For example, when a new environment is adapted, software engineers may add new functionality to benefit from its advantages. However, some researchers suggest that 17% of maintenance effort is devoted to corrective maintenance, 18% to adaptive maintenance, while perfective maintenance consumes 65% of the maintenance effort (Sommerville, 2011).

The major and first activity that leads to the higher cost proportion of software maintenance is the understanding process of the software system under maintenance as shown in Table 2.1. In particular, more than half of the maintenance costs are assigned to understanding the intended software system (IEEE CS, 2012). For example, software engineers have to spend a considerable amount of time required for maintenance process in understanding the software system and analyzing the impact of the proposed changes (Storey, 2005). Therefore, understanding process of an existing software system is a necessary prerequisite and costly activity, in particular, understanding of software systems that undergo several maintenance cycles is a difficult and a time-consuming task (Hamou-Lhadjand Lethbridge, 2010).

Table 2.1: Tasks and activities requiring program comprehension.

Maintenance Category	Activities
Corrective	Understanding the System ….
Adaptive	Understanding the System ….
Perfective	Understanding the System ….
Preventive	Understanding the System ….

The process of understanding or comprehending software systems is called program comprehension. Program comprehension has been a subject of extensive research studies for decades in order to reduce the costs and improve the quality of software maintenance. For example, Cornelissen *et al.* (2011) show the importance of trace analysis in performing adaptive and corrective maintenance tasks. However, maintaining an inadequately documented software system entails understanding of its various artifacts such as its source code and dynamic information. Inadequate is the level where the documentation is poor, out-of-date or at best insufficient. As a result, the problem of understanding how the system is implemented is a tedious, time-consuming and costly. The next section discusses program comprehension and its cognitive models.

2.3 Program Comprehension

Understanding what a program does (function), how the program works (implementation), and why the program is as is (design) is critical to software maintenance (Zhang, 2005). A large portion of the budget of software systems is devoted to the process of understanding and comprehending these issues. However, the understanding process varies greatly from a maintainer to another as it depends

mainly on the individual (Zaidman and Demeyer, 2008). Several factors can influence the understanding process such as the experience of the maintainer in the domain, the familiarity of the maintainer with the subject software system, the needed level of understanding, the programming language that implements the subject system and the magnitude of the subject system (Lakhotia, 1993; von Mayrhauser and Vans, 1995).

Several cognitive models and strategies have been presented for program comprehension. These cognitive models describe the cognitive processes and temporary information structures in the programmer's head that are used to form mental models (Storey, 2005). Mental models are sets of beliefs that a software engineer hold about how pieces of software, or software features, works. Cognitive models depend on strategies referred to as program comprehension models (Pennington, 1987b; von Mayrhauser and Vans, 1995; Storey *et al.*, 1997). In the literature, there are four accepted models and strategies of program comprehension, namely bottom-up model, top-down model, integrated model and partial model. The following subsections discuss these models in more details.

2.3.1 Bottom-up Model

This strategy assumes that software engineers first comprehend source code and then mentally chunk code statements into higher level of abstraction. These chunks are aggregated repeatedly until clear understanding of program is attained (Shniederman and Mayer, 1979; Pennington, 1987b; von Mayrhauser and Vans, 1995; Storey *et al.*, 1997). A chunk of code is usually consists of one or more than one basic blocks or it can be a part from a basic block.

Several research studies have used bottom-up strategy such as Shneiderman and Mayers (1979) and Pennington (1987a). For example, Pennington suggests that two kinds of mental models are needed namely, program model and situation model. A program model is a control flow abstraction that holds the behaviors of the program execution. Once the program model exists the situation model is mentally developed. A situation model is a data flow/functional abstraction. The development of the situation model requires the knowledge of real-world domain such as objects and events.

Overall, the bottom-up model begins with abstract concepts constructed by chunking code structure into higher level of abstraction. This strategy is used when the source code is totally new to the software engineer. Similarly, bottom-up strategy is used in understanding execution traces of method calls by means of exploring contents of various subtrees in the execution trace (Jerding and Rugaber, 1997). This research enables bottom-up strategy by exploring the interactions between individual classes in an execution trace.

2.3.2 Top-down Model

This strategy assumes that comprehension process starts from formulating general hypotheses about the purpose of the program. These general hypotheses are then refined into sub hypotheses as a hierarchical fashion. Sub hypotheses are evaluated and verified whether they are valid or not (Brooks, 1983). The verification of hypotheses depends heavily on the strength of the beacons in the source code (Brooks, 1983). If a hypothesis is invalid, a new hypotheses may be constructed and

verified. This process continues until an adequate understanding of the program is achieved.

Several research studies have used top-down strategy such as Brooks (1983) and Soloway and Ehrlich (1984). For example, Brooks (1983) assumes that software engineers create a mapping between the application domain and the programming domain in the development phase. Understanding process involves the reconstruction of this mapping through several intermediate domains. This reconstruction could be achieved by creation, confirmation, and refinement of hypotheses. However, Soloway and Ehrlich (1984) propose that understanding of a new code could be gained in a top-down model when the code is familiar.

Overall, the top-down model begins with a general hypothesis that leads to sub hypotheses. This strategy is used when the source code is familiar and when software engineers have some knowledge of the intended software system, for example reading documentation of the system. Similarly, a trace analysis consists of two steps: a) formulating hypotheses about the trace contents in term of what they do, and b) validating these hypotheses through matching them to the trace contents (Hamou-Lhadj, 2005). Whilst, the first step ought to be easy for maintainers who are familiar with the intended software system, the second step is not easy because of execution traces are very large in terms of size and complexity. The main contribution of this thesis is to enable top-down strategy through providing maintainers with simplified views of a particular trace at different levels of abstraction based on decoupling the tightly coupled modules. This contribution is discussed in more details in Chapter 6.

2.3.3 Integrated Model

This strategy combines the top-down and bottom-up models where software engineers use the suitable approaches while understanding the actual code (Letovsky, 1986; VonMayrhauser and Vans, 1993). For example, Von Mayrhauser and Vans (1993) present a meta-model to ensure that software engineers tend to switch among the different comprehension strategies depending on their expertise. This meta-model combines features of several existing models, particularly Soloway and Ehrlich's top-down model (Soloway and Ehrlich,1984) and Pennington's bottom-up model (Pennington, 1987a).Overall, the integrated model uses the different comprehension strategies to build understanding concurrently at several levels of abstractions by freely switching between the different comprehension strategies (Storey *et al.*, 1997). Similarly, this strategy could be applied in understanding execution traces by combining the bottom-up and top-down strategies where a software engineer can switch between them according to his needs.

2.3.4 Systematic and As-Needed Models

There are two approaches that software maintainers may use namely, systematic approach and as-needed approach (Littman et al., 1986). In particular, a systematic approach implies reading the code in detail and tracing through control and data flows (Littman et al., 1986). On the other hand, an as-needed approach implies focusing only on the related code (Littman et al., 1986). For example, Erdos and Sneed (1998) propose a partial comprehension strategy that assumes that there is no need to understand the whole program. Therefore, it localizes on a needed part of a program instead of understanding the whole program. The authors recommend that software maintenance tasks could be solved by answering a set of basic questions:

How does control flow reach a particular location?

Where is a particular subroutine or procedure invoked?

What are the arguments and results of a function?

Where is a particular variable set, used or queried?

Where is a particular variable declared?

What are the input and output of a particular module?

Where are data objects accessed?

However, some of these questions could be answered directly by analyzing traces of method calls such as the two first questions. The remaining questions could be answered by expanding traces of method calls to take into account arguments and return values or by providing another resource of understanding such as the source code itself.

Overall, as-needed or partial strategies are commonly used more than systematic strategy as the latter is less feasible for larger programs (Storey *et al.*, 1997). However, the former may overlook some important interactions that lead to more mistakes (Storey *et al.*, 1997). This research supportsthe partial strategy in order to gain its benefits. In particular, the contribution of this thesis is based on combination of three principles. The first of them is the scope filtering that aims at filtering the unwanted application modules and provides the capabilities to allow maintainers to determine their own wish to consider only wanted modules when performing the analysis. Hence, the analysis process is focused on the interested parts of the trace. The next section explores the key aspects of dynamic analysis such

as execution traces, its research directions, its main phases and its strengths and weaknesses.

2.4 Dynamic Analysis

Dynamic analysis is the investigation of the system behavior by analyzing its runtime information (Pirzadeh 2012). Runtime information is the information collected from the software system as it runs (Zaidman, 2006).This information illustrates distinct aspects of dynamic behavior of the investigated program such as control flow, data flow and event sequences (Zayour, 2002). Several contexts can benefit from dynamic analysis such as compilers, optimization, test coverage and program comprehension (Zaidman, 2006). In particular, dynamic analysis can help in understanding the functionality of a particular software system by examining its behavior (Ball, 1999).This research selects dynamic analysis to proceed with for two reasons:

1. Dynamic analysis can support as-needed comprehension strategy.
2. Dynamic analysis can precisely handle polymorphism in object-oriented systems as the wide use of polymorphism and late binding.

Regarding to the above first reason, as-needed strategy (or goal-oriented strategy) implies that only those interested parts of the subject software system should be analyzed. This strategy is useful to identify which parts exactly are related to a certain functionality of the subject software system. In addition, as-needed strategy is used frequently due to commercial pressures and time constraints. For example, when a software engineer has a little or even no knowledge of a certain software system, he/she needs only to execute specific scenarios related to the task at

hand. Consequently, the result of the analysis will be useful as the gathered information is more oriented. On the other hand, if a software engineer has to use a less goal-oriented strategy (i.e. static analysis), he/she should understand most parts of the subject software system before knowing which parts exactly are related to the needed functionality (Zaidman, 2006).

Secondly, Sintes (2002) defines polymorphism as: "polymorphism is the state of one having many forms. In programming terms, many forms mean that a single name can represent different codes selected among by some automatic mechanism. Thus polymorphism allows a single name to express many different behaviors". This leads to the notion of late binding where executed behaviors are determined at runtime. Although this mechanism is efficient in programming context, in contrast, it disturbs program comprehension process as it defers the precise behavior of the subject software system to the runtime. In particular, considering multiple possibilities of variations are difficult and time consuming task (Schach, 2010). Therefore, instead of considering all theoretical variations, dynamic analysis can determine the actual ones that are executed.

Regarding the dynamic analysis modes, they can either be online or offline, also known as ante-mortem and post-mortem modes respectively. The online analysis mode interleaves the analysis and recording of runtime information phases with program execution. However, the online analysis mode is considered as inefficient and time-consuming approach for program comprehension for several reasons. First, it prevents software maintainers from repeating the analysis without needing to execute the program in each time. Second, it slowdowns the program

execution, therefore, it is useful for limited cases when it is needed to monitor a certain part of the code.(Korel and Rilling, 1998; Ernst et al., 1999; Mock, 2003).

On the other hand, the offline analysis mode defers the analysis phase after the program execution terminates. Therefore, the recorded runtime information must be stored in a file called an execution trace file (refer to Section 2.4.2). The offline analysis mode has the advantage of enabling software maintainers to repeat analyzing the same runtime information several times without repeating the execution of the program each time. The following subsections discuss some features of the dynamic analysis.

2.4.1 Dynamic Analysis versus Static Analysis

The complementary technique for the dynamic analysis is static analysis, which uses the source code of a particular software system without executing it. Therefore, a static analysis uses source codes of programs as the main references in order to investigate their properties. Hence, a static analysis can help in understanding the static aspects of software systems such as their code structures. On the other hand, a static analysis cannot help in understanding the behaviors of software systems. In particular, the code structure of an object-oriented system often tends to be different from its runtime structure. For example, the code structure is usually frozen at compile time, whilst the runtime structure of the system consists of very complicated interwoven lattices of communicating objects. Hence, the code structure of the object-oriented system will not reveal the complete information about how the system will work (Muller*et al.,* 1993).