THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Investigating Strategies to Support Reverse-Engineering of Interactive Systems

Aman Alsharif

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

November 2013

Abstract

Most software applications today provide a graphical user interface (GUI), which facilitates the use of the software by offering graphical and visual elements to the users. The correctness of the user interface is fundamental to the correct implementation of the overall software. Using reverse engineering tools and methods is one of the most efficient ways to understand a system, and to assess its functionality and usability. However, traditional reverse engineering methods are not well suited to interactive elements of a system. This research examines and analyzes reverse engineering techniques of interactive systems written in the Java programming language, with the aim of creating a set of models of the entire system namely, *Presentation Models* (PModels) and *Presentation Interaction Models* (PIMs), which are very effective in describing structural and functional behavioral features of the interactive system. This study will also highlight some of the problems that exist in this domain and investigate several possibilities for improving the process.

Acknowledgments

First of all, I would like to express my gratitude to my supervisor Dr Judy Bowen, who not only helped and challenged me but also was very patient with me and provided beneficial comments, encouragement and reassurance during the course of completing this thesis. I appreciate all her advice and assistance during this project and in writing the report.

I would like to thank the librarian, MS. Lynch Helen, for her time and patience in editing and checking references list of this thesis.

Many thanks go to Alan Messenger and Jennifer Buckle for proofreading parts of this paper.

I am very grateful for the support and advice from all my friends in the Computer Science department, especially Hassan Yamani and Ali Al Harbi.

I would like to thank all my family for their unconditional love, praises and prayers for me. Very special thanks go to my emperor/ father, Abed Alsharif, and my empress/ mom, Fatimah Alsharif, for their constant efforts, encouragement, love and reassurance. Really, without them, I wouldn’t be who I am now. They have done so much that I couldn't repay no matter how hard I try. I would also like to thank all my brothers, Munther, Abdulbaset, Abduallah, Osama and Fouad. Sincere thanks are also to all the lovely friends and sisters who supported and encouraged me, especially the lovely princesses Noor, Sahar, Sanna, Basmah, Samiah, Radiya and Hameda; I doubt that I will ever be able to convey my appreciation fully.

Finally, all the thanks and appreciation goes to all the people, whose names I don't mention. They were unknown soldiers, who provided me timely support and encouragement during my studying here in New Zealand.

Contents

List of Figures

List of Tables

Chapter I

Introduction

Most modern operating systems offer graphical user interface (GUI) toolkits through which applications may present graphical and visual interactive elements to the users (e.g., menus, buttons, icons and windows), which enables users to interact with the application. Unfortunately, as systems and applications have grown in size and complexity, their GUIs have become less tractable and verifiable, and their users more prone to errors in input or understanding because of the increased complexity. Simply capturing user needs and intentions during specification or design phases is not enough to ensure that the implemented systems will support all of these needs and intentions in a usable manner.

Thus, the increase in complexity in the design of interactive systems leads to increased complexity of development of new software due to the difficulty of ensuring the correctness of interactive applications—i.e. their reliability, usability, effectiveness, and error-tolerance. Building software that is correct and robust is needed to ensure that the software does the correct thing in all the conditions. With safety-critical software in particular, the correctness requirements are considerable. Safety-critical software are those where any defect or failure in these systems could lead to severe harm in the lives of people, the environment or equipment. There are many examples of these systems, such as software in aeroplane cockpits, nuclear engineering, transport systems, interactive medical devices, weapons etc. These systems are safety-critical and financial losses and loss of life may result from their defect or failure.

There are, however plenty of usability evaluation methods (UEMs) that are used to develop the processes and manufacture of safety critical systems to minimise risk of implementation errors of these kind of systems and create correct system. Generally, these evaluation methods are divided into three groups: test, inspection and inquiry [Thimbleby06]. Test methods use representative users to work on typical tasks; testing helps to measure or evaluate users performances during interactions with the GUI (this method requires a working system, or a prototype). Inspection methods use expert evaluations to inspect a design; inspection can be carried at any phase of design, from prototype to marketplace phase. Inquiry methods investigate the preferences of users, and their desires and behaviors , and attempt to create the design requirements.

All these system evaluation methods focus on the technical and correct system design and are normally concerned with the reliability, integrity, effectiveness and safety of the system, where failure of such systems can directly present significant human life risk, cause to substantial economic loss, or lead to extensive damage of environment. Moreover, in safety-critical systems is not enough to feel confident that the system designs are precise and correct; we need proof and evidence of this even prior to implementation.

Testing GUIs and verification efforts are estimated at 50-60 per cent of total software development costs [Gray98] & [Perry95]. In Software Engineering, there are various techniques used to test the quality of correctness of the functionality of interactive system implementations. During testing, test cases are created and executed on the software and the results of the execution are compared against some test oracle, which defines the expected behavior . Test cases may either be created

manually by a tester, as described in works such as [Hicinbothom93] , [Walworth97], and [Foster98], or automatically by developing a specification-based testing technique to generate test cases [Chen01], [Memon01], [Paiva08].

A number of automated testing tools are available and follow a common pattern to test software, which is: producing test data, passing input data to the system under test, and recording results [Kuhn01]. The most popular approach typically used to support GUI testing is based on record/playback tools [Hicinbothom93]. The record tool captures mouse/keyboard events and all the GUI's screens as a user interacts with the system during the interactive session; later the test designer plays back these recorded sessions as needed to re-create the same events. This process can be costly and time consuming, and is prone to missing important decisions in GUIs due to the process of recording which requires user intervention to create the initial scripts [Memon01]. More essentially, these tools do not provide assistance to determine what tests are necessary and do not give any information about GUI functionality [Blackburn04] or coverage.

The correctness of an interactive application must also include an assessment of its usability—i.e. its effectiveness in satisfying user goals [Cortier07]. In other words, the application may be considered correct if it allows the user to perform the tasks for which it was designed. To design a usable system, one has to understand both its functionality and its intended users.

Thus, interactive systems design requires the embedding of knowledge, practices and experience from different views in order to achieve correct and usable systems. Designers need to check the properties about the design throughout the development process and include an analysis of human behavior in interacting with the UI, which

help provide an early verification to the designer before the application is actually implemented.

In order to verify the correctness of interactive systems, formal methods may be used. Formal methods are typically a mathematically based technique used to specify, develop, and verify software and hardware systems, using state tables or mathematical logic [Kuhn01]. The field of formal methods was developed to be more effective for finding system errors and omissions by providing formal specifications, as these can offer unambiguous, complete and concise models of interactive flow. Formal specifications describes the behavior of the intended system by using natural languages specifications such as Z [Jacky97], techniques such as theorem proving or model-checking [Kuhn01] "to ensure the specification is valid (i.e. meets the requirements and has been shown, perhaps by proof or other means of inspection, to have the properties the client requires of it) and a refinement process to transform the specification into an implementation" [Bowen08].

Formal methods are used until the system is built and tested, to ensure that we correctly understand what the requirements of the software are, that we design it in such a way that these requirements will always be met, and that we transform our designs into implementations that preserve these guarantees. By using these models, developers can identify possible problems that may need to be amended and rework the software systems where this prevents unexpected failures of the final system.

Whilst we typically consider a process where we apply formal methods during the design phase prior to implementing a system, formal methods can also prove useful in understanding of existing implementations. One way to achieve this is to apply the process "backwards", i.e. to reverse-engineer and analyze the code of these systems to

a more abstract specification. This, in turn, aims at promoting better understanding of the content or processes of the system and learning the principles that are used to design correct systems. In fact, analyzing and understanding legacy applications greatly reduces the amount of work required to improve an effective software system and reduces the need to develop a major part of the program from scratch [Huntington02]. Typically, software engineers work with reverse-engineering of existing or legacy systems, and analyze them to represent the results in an understandable way. This representation can be used to produce substitutes, upgrades or improvements of legacy systems, and can be used to check the required properties of the system.

Accordingly, one of the most efficient ways to understand a system and assess its behaviors is through reverse engineering, which we might define as the process of analyzing available software artifacts, such as requirements, design, architectures, code or byte-code, with the objective of extracting information and providing high-level views of the underlying system [Sommerville04] . Thus, reverse engineering can serve as a starting point for understanding the system , and constructing models of its behaviors .

Unfortunately, traditional reverse engineering is not well suited to analysis of the interactive elements of a system. This is because traditional reverse engineering cannot be used to discover design flaws [Gimblett10], is time consuming, and sometimes needs input from the original developer of the system to solve problems during the process [Belmabrouk12]. Moreover, GUIs have different characteristics from those of traditional software, and thus conventional reverse engineering methods are not appropriate for GUI systems analysis. An interactive application is composed

of (1) a user interface, which a user interacts with, and (2) the system functionality, which is the underlying behavior of the system [Lin12]; therefore we need a reverse-engineering process, which can capture all details of both of these components. The interactive elements and their relationship to underlying system behavior are difficult to capture, as we will discuss in more detail later.

Several recent works have proposed new reverse engineering techniques for GUI testing purposes. These involve the creation of some kind of visual and formal model of GUI behavior, for understanding the structure and execution behaviors of the interactive system. Some of these techniques analyze source code statically to derive UI elements [Silva06], and some use dynamic reverse engineering such as [Memon03], which uses the GUI Ripper tool that runs the interactive system and automatically captures all information about its windows, properties and values. Others, such as [Paiva08] attempt to capture the user functionality of the system for model-based testing purposes, while still other approaches have tried to describe functionality and behavior of actual interactive devices [Gimblett10].

These studies succeed in building formal models of GUI behavior, for understanding the structure and behaviors of the interactive system. However, most of them describe the user interface of the interactive system and ignore the underlying system specification. This disregard, unfortunately, is not conducive to a full analysis of the system; it causes difficulty in proving properties about the whole system to ensure that the software does the correct thing in all conditions, which is needed to build precise and correct system designs. Formal methods might be the only way to demonstrate correct functions of the system by describing both the structure and functionality of interactive systems, which is not the usual approach taken.

This study aims to find ways of using reverse engineering of interactive systems from legacy code in order to extract structural and functional aspects of the underlying system. We want reverse engineering of interactive system to produce formal models, which in turn, can be used for analysis and test derivation for the system. Our goal is to produce two particular types of model: *Presentation Models* (of structure and functionality behaviors) and *Presentation Interaction Models* (of interactive behavior), both developed by Bowen and Reeves [Bowen06] and [Bowen07] for constructing formal descriptions of interactive systems.

1.1 Research Purpose

This paper presents an investigation into reverse-engineering techniques for interactive systems written in the Java programming language, with the aim of creating a set of formal models for entire systems, which are *Presentation Models* (PModels) and *Presentation Interaction Models* (PIMs). It highlights some of the problems that exist in this domain and investigates several possibilities for improving the process.

1.2 Study Scope

Four different approaches had to be investigated before arriving at a satisfactory result; where, the scope of this study was to examine and analyze reverse engineering techniques for interactive software applications from legacy code. However, the existing reverse engineering techniques produce very big models due to the large amount of information in typical systems. This makes the analysis process slow and resource exhaustive.

For this purpose, the original approach was using detection techniques to simplify and decrease the effort in the analysis process. This attempt relied on Lin's (2012) study [Lin12], which described how the program-slicing techniques can be combined with a program dependency graph (PDG) of the system, which assists to create PModels and PIMs; yet our approach used clone detection techniques rather than program-slicing techniques. This method examined three different experiments for reducing information and understanding the complexity of source codes. Firstly, using the clone fragments only to be analyzed; secondly, removing all the clone fragments from source code and keeping just one copy of each set; and thirdly, removing all the clone fragments by using specific heuristic procedures to keep the information that is responsible for creating the models. Section 5.1 describes this in more detail. However, because this approach did not prove to be very useful, the scope was extended many times.

The approach was then extended to the use of program dependence graphs (PDGs) to extract widgets and their functionality and behaviors from the Java code of an interactive program, which in turn contribute to obtain the required models. This approach was different from [Lin12] in terms of using an automatic tool to generate the PDGs. The reason for that was to simplify the generating process of PDGS to examine and handle more than one example of interactive applications and address all the possible problems. Section 5.2 describes this in more detail. However, this experiment identified that missing information prevents the construction of the full models of interactive systems.

Therefore, the approach was further extended to use one of the dynamic reverse engineering methods to investigate whether the final information from both dynamic

and static methods can help each other to create the required models of the interactive system. This is explained in detail in section 5.3. This approach also was unsatisfactory as it also led to incomplete models.

The next approach was of analyzing the direct static source code of UI programs by reading the source code line by line to discover the parts that belong to a GUI system, and then extracting its widgets with their functionalities and the relationship between these widgets to build our models, chapter 5.4 gives more explanation about this. Unfortunately, this approach had some limitations when dealing with advanced and complex applications.

The fifth and final approach produced a satisfactory result. This approach is based on extracting the program entities from code via a parser tree and then traversing that tree and checking these entities to extract information required to build the PModels and PIMs. This is described in detail in chapter 6 and 7.

1.3 Report Outline

The rest of this paper is organized as follows:

• Chapter 2 provides some background knowledge about reverse engineering and the formal models used in this study, supported by examples, and also gives a brief background of clone detection techniques. Next, chapter 3 presents the related work in reverse engineering techniques using both static and dynamic methods.

• Chapter 4 explores all the software examples used through this research, and discusses the reasons for using them. Follow that chapter 5, which offers an overview of the initial investigations conducted in this paper, and the problems faced in each

experiment. The final section in this chapter gives a summary of what we have learned from these different experiences.

• Chapter 6 describes our analysis approach of static reverse engineering to extract the particular set of models from parser tree using tree-walking methods. This chapter gives an overview of the ANTLR tool that is used to automatically generate the parser trees of an interactive system and the tree-walking mechanisms. Chapter 7 describes how to extract GUI models from source code through the parser tree. Chapter 8 gives an overview solution of some problems in a complex interactive example by using this approach. We then present conclusions and discussions for future work in Chapter 9.

Chapter II

Background

Our goal in the first chapter of this study is to use reverse engineering of interactive systems from the existing code to identify all the components and their actions in the graphical user interface in order to understand the structural and functionality behaviors of the interactive systems. First, we are going to provide a general overview of reverse engineering, and discusses the different approaches that can be used for deriving required information from legacy code, and to describe the formal models and the set of models used throughout the paper. Next, we introduce the background to the clone detection techniques that are used as one of the suggestions offered during our investigations and experiments.

2.1 Reverse Engineering

Reverse engineering supports development of descriptions of software from high levels of abstraction (i.e. architecture), down to relatively low levels of abstraction (i.e. source code). It is hard to understand a system's structure at this lower level of abstraction [Muller09]. Chikofsky and Cross define reverse engineering as follows:

> "Reverse engineering is the process of analysing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of abstraction" [Chikofsky90].

Reverse engineering can provide a number of benefits, including improved documentation, maintenance, system evolution and reuse, specification-based testing, and re-engineering support. Reverse engineering is usually offered as a design model

for building a better understanding of a system, and can be used in code generation processes for updating legacy systems, or creating new systems with different features and functionalities. It can also be used to prove functional completion of system requirements [Systä00].

There are two basic types of reverse engineering: static, which extracts information from source code to describe the structure of the software, and dynamic, which derives information from a program during execution to describe its run-time behavior (typically "simulating the actions of a user exploring the system's state space") [Systä00], [Muller09]. Both approaches consist of the same three main phases: (1) data extraction, (2) data analysis, and (3) data representation/visualization, which represents the analysis results in an understandable way [Doan08], [Muller09], [O'Brien05], [Pacione03].

In reverse engineering, there are three types of views that can be used to clarify extraction of data from code: (1) static views, (2) dynamic views, and (3) merged views, which combine both static and dynamic information into a single view [Systä00]. These views are illustrated in Figure 2.1.

The static information can be represented in several ways, e.g. an abstract syntax tree (AST), a program dependency graph (PDG), class diagram, etc. Static information about software typically consists of software artifacts and their relations. In Java, for example, artifacts might include classes, interfaces, methods, and/or variables, whereas relations might include extensions among classes or interfaces, method calls between methods, and so on.

Dynamic information contains software artifacts as well, and may also include information about sequential events, concurrency, memory management and leaks, code coverage, etc. ([Systä00] and [Systä99]). This can be extracted using debuggers, event recorders, and/or general tracer tools [Martinez11].

Finally, a single view would directly clarify connections between both static and dynamic information views. If the static information is extracted from the source code some of the artifacts and relations might have been ignored. One example of such an artifact is a default constructor that is not explicitly provided in the source code. Since default constructors are invoked when an instance of a class is created, dynamic analysis can capture them. Constructing abstractions for single views can be difficult because the abstractions for dynamic and static views are usually significantly different. Dynamic abstractions are normally behavioral patterns or use cases, while the static abstractions are subsystems.

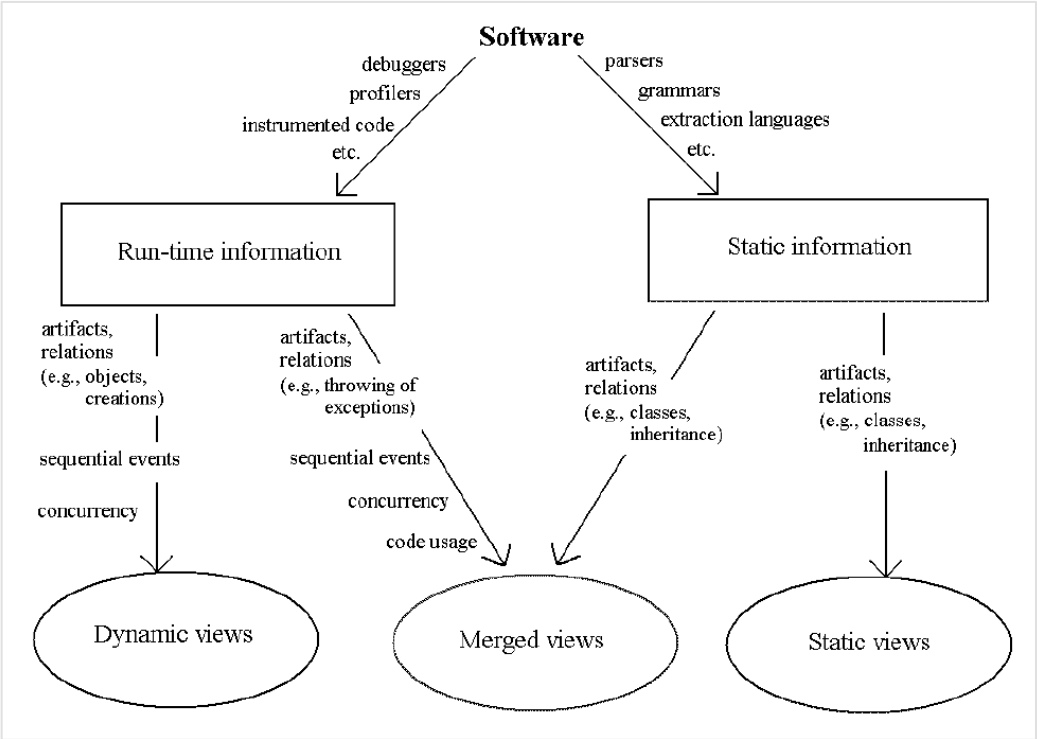


Figure 2.1. The different choices of view for software reverse engineering (Adapted from [Systä00])

During this study, our analysis uses the above types of views to derive the information needed to build particular formal models, namely, the PModels and PIMs proposed by Bowen & Reeves in [Bowen07], [Bowen06]. The next section provides an overview of formal methods and the PModels and PIMs in detail.

2.2 Formal Methods

In general, formal methods are used to predict and understand the functionality and behavior of systems, and to design and implement more correct, robust, and usable systems. For certain purposes, formal methods can describe the behavior of the program more concisely and with greater clarity than code. When applied to user interface systems, formal models can help to ensure consistency across target platforms, establish reachability and completeness, and, perhaps most importantly, incorporate user interface design into larger, formally constrained software development processes. Often, descriptions of formal models are presented in isolation from real-world contexts [Bowen07]. According to Gimblett and Thimbleby processes that rely on formal methods can guarantee a comprehensive analysis, yet are seldom used outside of certain key areas, presumably due to the high levels of experience these methods require and also these techniques take a lot of effort before implementation [Gimblett10]. Thus, formal methods sometimes are only applied to a model prior to implementation, such that "the implementation itself may have unknown bugs independent of the model" [Gimblett10].

Although formal methods give high priority to abstract properties of the software systems, they ignore issues of GUIs, such as number representation, line and page format, or output medium [Nau82, p.441]. There are few efforts at formally describing GUI issues that cover mostly understanding the structure of the interactive system (e.g. [Memon03], [Silva06]). While these approaches can offer useful insights into interactive systems, they are not capable of supporting a full system-level analysis, where capturing and analyzing the underlying system behavior of interactive systems as well their structure properties are required.

In order to attempt analyzing interactive systems and capture the interactive elements and their relationship to underlying system behavior with the powerful verification capabilities of formal methods, this paper uses both *Presentation models* (PModels) and *presentation interaction models* (PIMs). Although detailed descriptions of these models are provided in [Bowen07] and [Bowen06], brief overviews of these models follow.

Presentation Models (PModels)

The purpose of using a Presentation model (PModel) is to "formally capture the meaning of an informal design artifact, such as a scenario, storyboard, or prototype" [Bowen07]. The PModel is kept simple to match the simplicity of the informal artifacts it describes.

PModels can also be used to describe the features of the GUIs of implemented interactive software systems. By using PModels, all relevant behavior of GUIs and GUI designs can be captured, such as display and layout properties of the widgets [Bowen12].

As an added benefit, its simplicity also facilitates its wider adoption [Bowen06]. The "*meaning*" of a design is intended to identify the behavior of an informal design to remove any ambiguity of the design that enables designers to consider it during the design process. PModels describe the behavior of a UI (either from a design or implementation) in terms of its component windows (i.e. "widgets", such as buttons, menu items, text entries etc.). Each widget description has three properties: a name, a category and a set of behaviors. The name is used to identify the component; the category provides the component's classification (e.g., buttons are 'ActionControls',

radio buttons are Selectors, etc.); and the behaviors indicate the various actions and events associated with the widget.

Behaviors are divided into two types: interaction behaviors (I-Behaviors) and system behaviors (S-Behaviors). I-Behaviors are behaviors that affect the user interface in some way, whether through navigation from one part of the UI to another, or through changes to some feature of the UI. This means that any I-behavior determines the user interface's reaction to the users' actions. S-Behaviors are behaviors that affect the underlying functionality of the system. They affect the underlying state of the system [Bowen07].

To illustrate, this paper uses a simplified example of a "BMI Calculator", a small application written by the author using the Java language and the Swing GUI library. This small example contains enough detail to explain the concepts of the models described above but is simple enough to be easily described and understood. The application consists of two windows/dialogues, which contain a variety of interactive widgets. When the application is started, users need to fill in the height and weight and other relative information, then click on the button "calculate" to produce a Body Mass Index (BMI) in another window, as depicted in Figure 2.2.

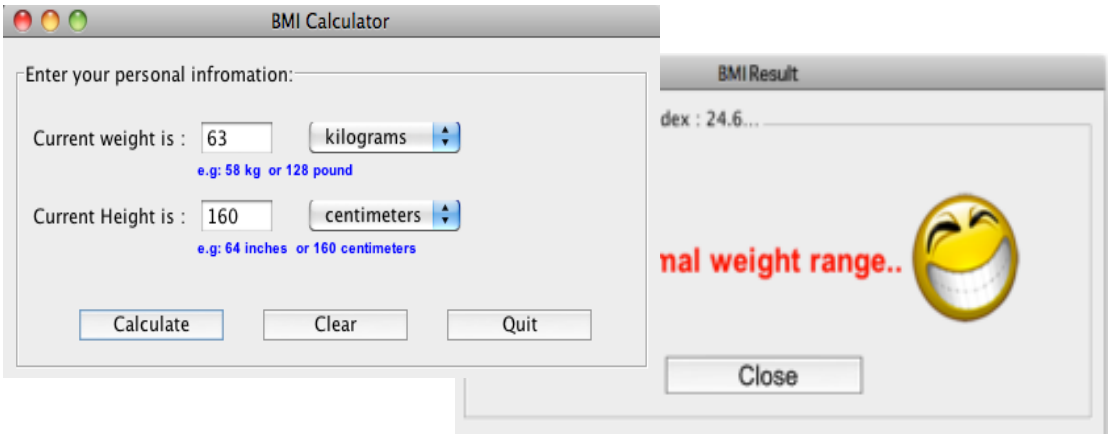


Figure 2.2. Screenshots for the "BMI Calculator" windows.

The presentation model for this example is denoted by:

```
BMICalculater is MainWin : ResultWin
MainWin is
   (WeightEntry, Entry, {})
   (WeightEntry, SvalueResponder, (S_clear))
   (WeightSel, Container, {})
         (KilogramsItem, SvalSelector,{})
         (PoundsItem, SvalSelector,{})
   (HeightEntry, Entry, {})
   (HeightEntry, SvalueResponder, (S_clear))
   (HeightSel, Container,{})
          (CentimeteressItem, SvalSelector,{})
          (InchesItem, SvalSelector,{})
   (CaculateButton, ActionControl, (S_result, I_Calculate))
   (ClearButton, ActionControl, (S_Cleare))
   (QuitButton, ActionControl, ((QuitApp))
ResultWin is
   (Result, SvalueResponder, (S_result)
   (CloseButton, ActionControl, (I_Close))
```

This PModel describes a UI such as that given in figure 2.2. It has two different windows, which are: 'MainWin' and 'ResultWin'. Each window has its own presentation model and within these each widget is described. ' MainWin' for example has seven widgets, including three buttons: (1) 'CalculateButton', (2) 'ClearButton' and (3) 'QuitButton', which are all ActionControls. The behaviors associated with 'CalculateButton' are 'I_Calculate' and 'S_result'; in other words, this widget has two different behaviors, a system behavior, which refers to mathematical calculations within the system, and an interaction behavior, which means that a new window will appear to display the result when the user interacts with this widget. For widget 'ClearButton' the associated behavior is 'S_Clear', which means that there is some function of the system to clear some widgets' values in the application, such as: Weight Textfiled. This widget is categorized in this case as 'SvalueResponder' and its behavior 'S_Clear' that signifies the behavior that affects this widget responder. Also, this widget is described as '*entry'*, where a user can enter particular information. Finally for widget 'QuitButton', the associated behavior is 'QuitApp', which means

the behavior of this widget is to quit the system. Thus, the PModel gives a description of each window and describes all the widgets into this window with their details and behaviors. To clarify, Figure 2.3 shows the description of the PModels for the main window, “BMI Calculator” in the application, and it also describes some widgets in this window.

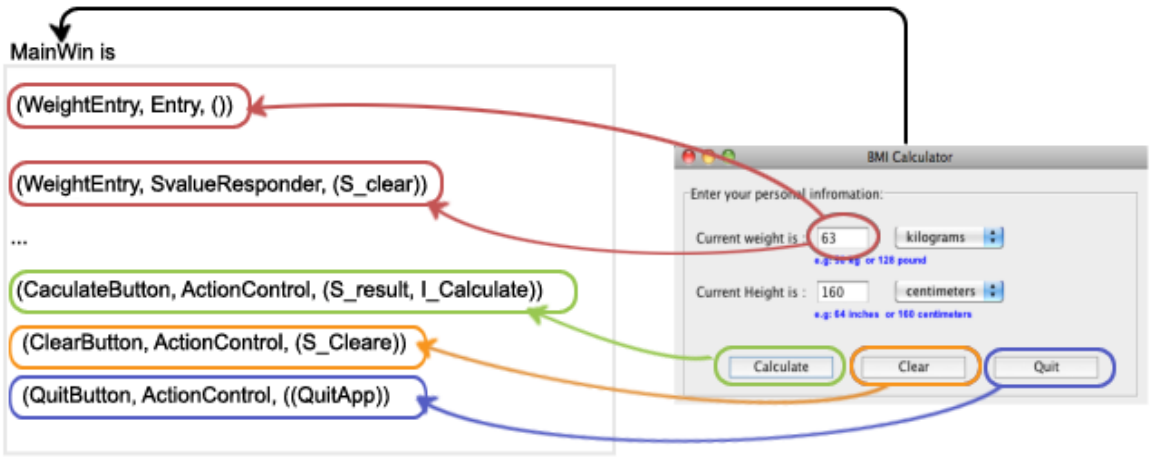


Figure 2.3. The representation PModels of the main window of “BMI Calculator” app with some widgets.

Presentation interaction models (PIMs)

A PIM is used to describe the possibilities for navigation and state transition in the UI. It is a finite state automata and is typically described using the μChart language [Reeve05]. The PIM consists of (1) states, represented by ovals, including an initial state distinguished by a double ring, and (2) transitions, represented by arrowed lines. The transitions are described by a trigger and an action, separated by a "/". The trigger is a boolean expression over the input set. [Reeve05].

The PIM is derived from the presentation model, in which each state represents a distinct window or dialogue of the GUI [Bowen08], and transitions represent the movements between these states. A transition describes the changes resulting from I-behaviors available in that state, (indicated by the guard name which is an I-behavior). The PIM for the "BMI Calculator" example is given in Figure 2.4. Note that this PIM provides a clear understanding of all relevant state transitions in the application.

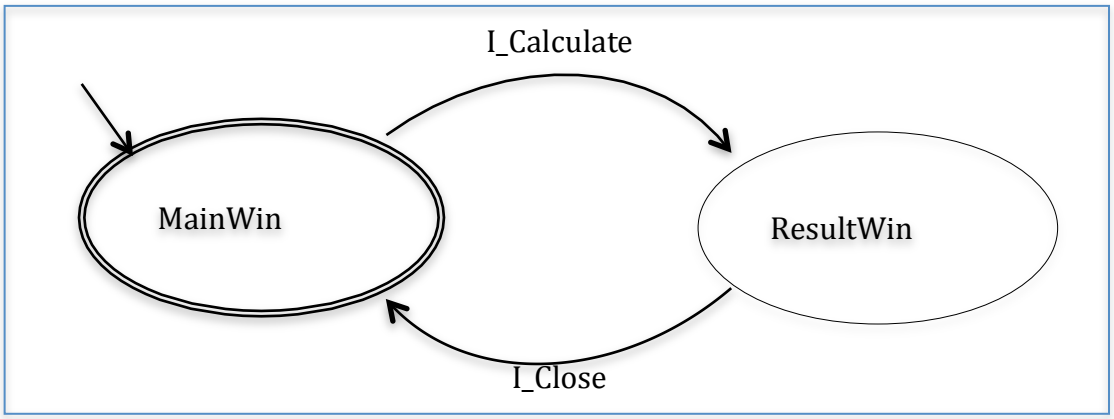


Figure 2.4. PIM for "BMI Calculator"

In Figure 2.4, where each component of the presentation model (e.g. MainWin and ResultWin) is shown as a state, and transitions between states are labelled by an

I_behavior, which represents such a state change. Thus, a PIM provides an understanding of the movements and changes between these states.

PModels and PIMs are offered to analyze the interactive system and capture the structure properties and underlying system behaviors of these systems. Thus, they are the target models for our reverse-engineering approaches that are we want to be able to gather enough information from the reverse-engineering process to enable us to create these models. We discuss this in more detail in chapters 5 and 6.

2.3 Code clone detection

A number of existing studies ([Lague97, Baker95], [Davey95), [Mishne04]); show that code clones or copying of a code fragment may account for between (5-30 per cent) of the total amount of code in large software systems. However, there isn't any specific definition for a 'code clone', and all previous works suggest their own definition [Higo07]. According to Muller et al. (2009) clone is "Source code segments that are structurally or syntactically similar"[Muller09], whereas Baxter et al. defines the clone as "A program fragment that is identical to another fragment" Baxter98]. Near miss clones are two or more fragments in code identical to the other [Baxter98].

The clone has a number of types, which are determined according to the two main kinds of similarity between code fragments. Fragments could be similar based on: (1) The textual similarities, which are often due to copy-and-paste processes, or (2) Based on functional similarities, which can be independent of their text. Thus, the types of clone based on these two kinds of similarities are: [Roy07], [Roy09], [Davey95]

- Textual Similarity: Based on the similarity of program text, the clone types are divided into:

  Type-1: The two code fragments are similar to each other except for some variations in white space, layout and comments.

  Type-2: The syntactic structure of the two code segments is similar except for variations in identifiers, types, literals, whitespace, layout and comments.

  Type-3: There are other modifications in copied fragments such as changed, added or removed statements, alongside variations in identifiers, literals, types, whitespace, layout and comments.

- Functional Similarity: Here there is one type based on the similarity of function, which is:

    Type-4: All the code fragments have the same semantics and functionality, which implement the same computation but are performed by different syntactic variants.

Typically, clones might be the result of the process of copying and pasting during programming [Roy09], optionally editing, producing exact or near miss clones [Baxter98]. There are code fragments that are merely incidental accidentally similar, which are not clones. Clones can also be presented by accidents, and may be created without knowing in the software systems. This is common when using APIs and libraries, which normally needs a sequence of procedure calls and/or other ordered series of commands. For example when creating a button using the Java Swing GUI library, this widget has a block of command code, which will be repeated for each instantiation [Roy07].

Figure 2.5 summarizes all of the various factors and reasons for using duplicated codes in the system (as proposed by, Roy et al., in [Roy07]). In fact, cloning is very common and useful in many ways. According to Baker [Baker95] programmers duplicate code for many reasons:

1) Cloning a part of a code is easier and faster than introducing a new code, where the code fragments may already be tested.
2) Copying helps to reduce the high cost of a procedure call for efficiency considerations.

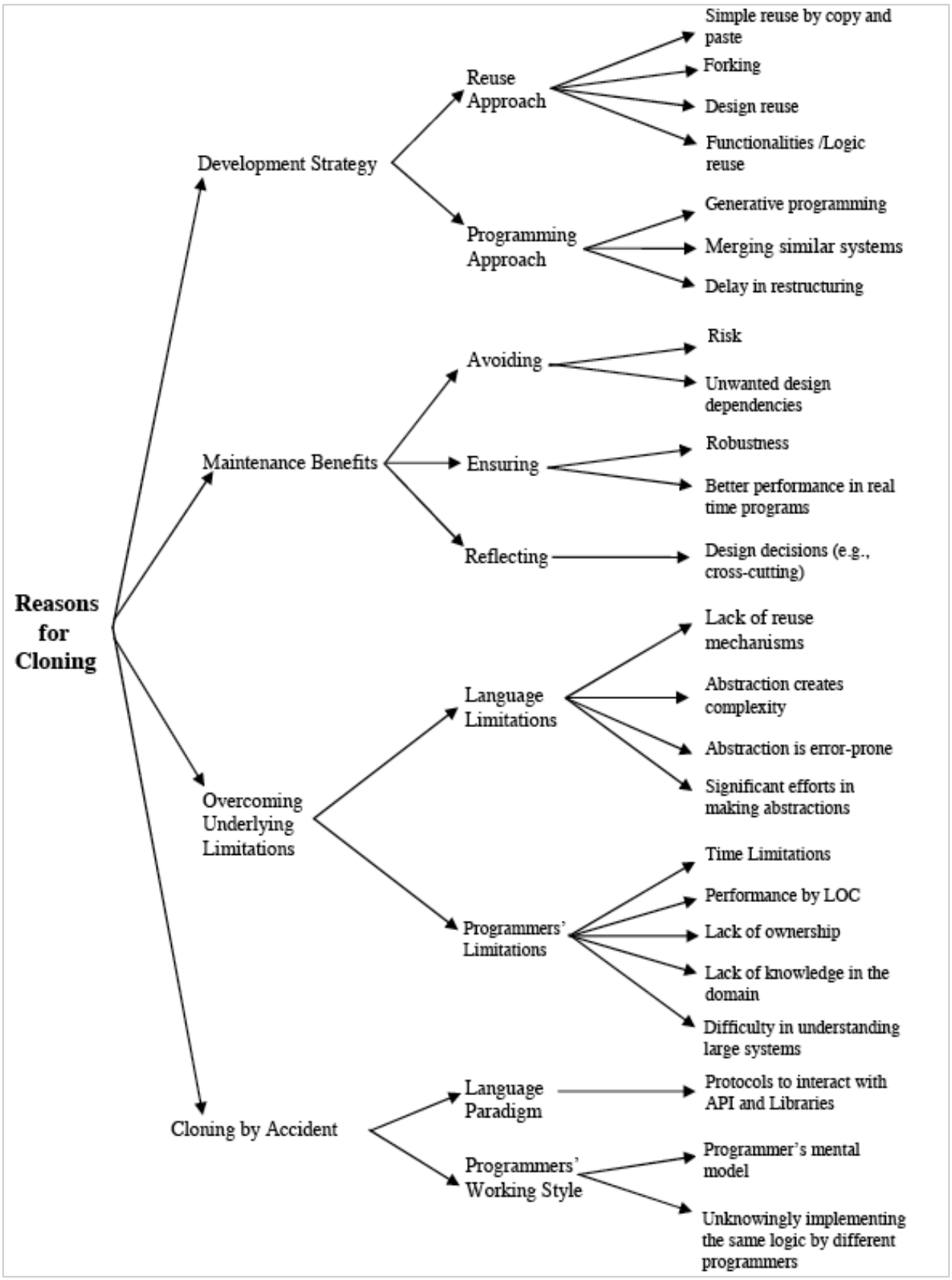


Figure 2.5. Tree-diagram shows the various factors of cloning in the source code (Adapted from [Roy07])

However, code duplication can cause maintenance and evolution drawbacks, because (1) repeating of errors in one clone generates these errors in many clones, and thereby (2) modification of these errors to a clone is required to occur to all of its clones, and also (3) too much cloning makes the code larger and more complex and often shows design problems [Roy07], [Roy09], [Muller09], [Evans07], [Baker95]. This forces programmers to examine code more than is necessary, and thus increases the cost of software maintenance [Baxter98].

Although the cost of maintaining clones over a system's lifetime has not been estimated yet, it is at least agreed that the financial impact on maintenance is very high. The costs of changes carried out after delivery is estimated at 40 per cent - 70 per cent of the total costs during a system's lifetime [Roy07]. Due to the large amount of cloned code and its maintenance cost, it is essential to detect code clones of large systems.

In reverse engineering, there are several reasons for the use of clone code detection technique. Firstly, it is used to remove bugs, where detecting and then removing an unnecessary duplicated code helps to eliminate a bug from the system only once, rather than repeating the removal process separately for each cloned part. Secondly, clone detection helps to reduce code bloat, where restructuring cloned codes into one function reduces the source code size and later also that of the executable. Finally, this technique helps to repair design-flaws such as missing use of inheritance, where these problems could be the result of a code duplication process [Rieger98].

From the above description, we infer that using a clone detection technique could be helpful in our examination and analysis of the reverse engineering techniques of any interactive system to generate a set of models; especially where during the process of

reverse engineering, there may be difficulty in analyzing the code due to the complexity of source code in large systems. Clone detection techniques are used to reduce the information of code, and then this may help to decrease the effort required in the analysis process. Thus, one of the suggestions that we introduce in this study is to investigate whether the use of clone detection can assist with this reverse engineering process. This is described in detail in Section 5.1.

Chapter III

Related Works

3.1 Introduction

According to Issa, Sillito and Garousi [Issa11], graphical user interface based software systems represent more than 60 per cent of software systems under development today. As argued by Paiva, Grilo and Faria [Paiva10], graphical user interfaces are commonly used in software acting as mediators between systems and the users of those systems. Their quality affects the decision of a user to use them. One method of reducing the effort required in the construction of graphical user interfaces is through the production of partial models from legacy code by a reverse engineering process, which involves the extraction of information on the structure and behavior of a graphical user interface through the combination of automatic and manual exploration. This reverse engineering process starts by building a preliminary application model through interaction with the existing graphical user interface. The model obtained through reverse engineering captures information on the structure of the graphical user interface.

This section reviews several studies related to reverse engineering techniques using both static and dynamic techniques.

A common technique used to discover system deficiencies is GUI testing. Several studies have used reverse engineering techniques for the purpose of testing graphical user interfaces by extracting models to be used in a specification-based testing process. The purpose of using these techniques is to extract structural and execution behaviors of the interactive systems.

As shown by Silva, Saraiva and Campos [Silva10], the user interface layer has a high probability of being changed during the lifetime of an application. However, available technologies, used mostly to construct user interfaces, comprise component libraries grouped together in event styled programming making the program code difficult to understand and maintain. Swing[1] components are based on the model view controller architecture pattern; this does not imply that the pattern is maintained at the application level [Silva10]. Model-based designs are used to help designers to specify and analyze systems. Integrated development environments facilitate the creation of user interfaces but do not promote better code structures and quality. Reverse engineering tools help in analyzing, understanding and manipulating source code.

Strategic programming uses predefined sets of generic transversal functions to traverse any Abstract Syntax tree (AST) using different traversal strategies such as left to right and top down. Functions enable us to concentrate on nodes of interest only. Abstract behavioral and structural models can be obtained through the use of code slicing techniques and by examining the abstract syntax tree (AST) of a Java program. The slicing programming technique is applied so as to ignore irrelevant information from the system and focus on the Swing sub-program from the whole Java program. Strategic programming extracts the Swing fragment from a Java program and can be reused to slice another GUI tool kit for other languages / ASTs. The final extraction models generate interaction models and event flow graphs.

[1] The Java swing library is the primary Java GUI widget toolkit that provides GUI for Java programs.

3.2 Static Analysis

The static method is one type of reverse engineering technique. It is based on deriving the information from source code to describe a system. This extraction can be by using methods such as parsers, grammars, extraction languages, and so on, to present the information as parser tree, ASTs, PDGs, etc. for easy analysis. Many studies use the static reverse engineering method to analyse the existing code [Silva11], [Staiger07], and [Cortier07].

Silva, Saraiva and Campos [Silva11] and [Silva06] describe how the abstraction of a graphical user interface can be identified from legacy code by detecting user interface components through functional strategies and formal methods. These components include user interface objects and actions. Slicing functions are constructed to isolate the Swing sub-program from the entire Java program. The straightforward approach involves defining an explicit recursive function, which traverses the abstract syntax tree (AST) of the Java program and returns the Swing sub-tree. They simplify the usage of user software through the provision of visual controls. Their approach was able to extract the application's windows, and all their widgets, properties, and values. At the end, the output of this process is used to create three models: (1) event-flow graphs, which show the all-graphical user interface elements and their relationships; (2) interactor based models, which capture the interaction perspective of the GUIs; and finally (3) finite state machines.

In another study, Silva, Saraiva and Campos [Silva10] produced GUI Surfer, a tool that reverse engineers the GUI layer of interactive computing systems with the major aim of enabling analysis of interactive system from source code. Model-based software development, specifically of interactive computing systems, uses models to

guide the development process. These are iteratively refined until the system source code is obtained. Models provide useful information for system maintenance and development. The GUI Surfer tool uses several techniques to simplify the achievement of GUI source code reverse engineering, which can be easily retargeted. A parser is used on the appropriate programming language to obtain the abstract syntax tree from the source code. The construction of a function that isolates a portion of the program from the entire program facilitates the extraction of the user interface from the abstract syntax tree (AST). Generic techniques that work in any AST are used since the approach is intended to be used across many different programming languages and paradigms. The reverse engineering approach thus combines two language independent techniques: strategic programming and program slicing. The components of the GUI constructors are used to focus the slicing in the sub-trees, which represent the graphical user interface. The GUI code-slicing module slices code of relevant GUI AST fragments and performs a tree transversal based on the program dependency graph to detect all GUI nodes. A generic model can be extracted from the GUI from any AST such as C#, WxHaskell, Java / Swing.

The use of the GUI Surfer tool is aimed at simplifying the manipulation of the AST, thus making it easily re-targetable to different programming languages and GUI tool kits. From the user interface code of interactive systems and a list of relevant components of the graphical user interface, a graphical user interface abstraction is generated. Models can be extracted at different levels of detail since the GUI Surfer receives the lists of components as a parameter; where interactor based models capture a user-oriented view of the interface, and event flow graphs capture the internal structure of the code.

Another interesting study in this area has been conducted by Staiger [Staiger07]. It proposes an approach for analyzing graphical user interface programs statically detecting GUI program parts and widgets with the hierarchies they form showing event handlers connected to events of those widgets. The study focuses on programs written in C or C++, which use GUI libraries such as GTK or Qt. In addition to supporting the general program understanding, this also supports the analysis of the flow of control, which enables architecture recovery, migration to GUI constructors and mapping the visual appearance of the program to source code artifacts. Static analysis of interactive systems can also be used to analyze applications. It begins by finding program entities, which belong to the graphical user interface, which facilitates recovery. The next step involves finding the widgets and the hierarchies they form so as to recover program windows' structure. This enables the mapping of the program visual appearance to source code artifacts and vice versa and migration to GUI builders. The final step involves searching for widget emitted events and for the functions connected to them as reactions. This facilitates the analysis of the flow of control and facilities the examination of the behavior of the program in source code. The Bauhaus tool suite describes static GUI analysis [Staiger07]. The reaction to user actions is managed through events, which operate in the same way as function pointers. Static program analysis encounters several problems when they are applied to problems with a graphical user interface. To begin with, the analysis only sees the code of the client and not the GUI library implementation. The large size of graphical user interfaces also complicates the analysis, which means that the analysis has to be efficient.

Cortier, d'Ausbourg and Aït-Ameur, [Cortier07] performed a study to investigate the applicability of reverse engineering and formal approaches to the validation of UI

correctness. Their technique uses the static analysis of Java / Swing applications. It involves deriving a user interface from its Java/Swing code after which the formal execution model is used to prove that the developed interactive system fulfills the usability requirements expressed in Concur Task Tree (CTT) models [Cortier07]. Finally, these models extracted the widgets and listeners instantiations and initializations of the UI system. The CTT model formalizes in an Event B models that simulates the effective reactions of the encoded application in response to the user actions. The UI structure and behaviors are captured by modeling the Swing library and by abstracting listener methods in a set of B events scheduled by variants; where all the widgets in Java application commonly are defined in the specific API libraries, e.g. Swing and AWT libraries; and the actions of these elements are defined in the implementation code of the listener methods.

Lin's study [Lin12] discussed reverse engineering for interactive systems to extract the user interface's structure from programs written in Java using the Swing GUI library. This study tries to analyze the source code statically by using program slicing techniques combined with program dependence graphs of the system (PDGs) to extract behavioral and structural models. The slicing technique is used to decrease needless source code and obtain the relevant source code for creating the required models of GUIs systems. The final aim was for the extraction of the necessary data to generate PModels and PIMs. This work focused on principles and theories for extracting information and described a manual process to achieve this.

3.3 Dynamic Analysis

Another alternative is the use of dynamic analysis technique, which relies on extracting information from a program during execution, by using debuggers,

profilers, event recorders, etc. to describe its run-time behavior. One of the most well-known dynamic analysis approaches for interactive systems is GUIRipper, which was developed by Memon et al. [Memon03]. Their studies use dynamic analysis to extract behavior and GUI structure models. The GUI Ripper tool can extract all information about GUI widgets, properties (such as the background color and font), and values of these properties (such as red, Times New Roman, 18 pt.), by opening all the windows under the test. A GUI model is generated. This represents the GUI structure as a GUI forest and its execution behavior is represented as an event-flow graph and integration tree. The GUI windows are represented in the GUI forest as the nodes of the forest while the hierarchical relationship between the windows is represented as edges. Every node clarifies the state of a window, which constitutes the window's widgets, their properties and values, while the event-flow graphs represent all the possible interactions among the events of a GUI component. The major problem with this tool for our purposes is that since the extraction of the GUI model from the GUI application is automated fully, some GUI windows (and therefore some parts of the UI) may be missed by the ripper due to the requirement of user interaction such as entry (passwords or data) to fully navigate the system, or in cases where widgets which quit the application are invoked as part of the exploration process.

Another popular technique designed for the purposes of GUI testing was presented by Paiva et al. [Paiva08]. They used a dynamic GUI reverse engineering process with the GUI reverse engineering tool, ReGUI2FM. The aim of their study is to minimize the effort required in building a GUI model for the purposes of model-based testing. The tool REGUI2FM extracts structural and behavioral information about the graphical user interface being tested. It uses a dynamic exploration process, which mixes automatic and manual exploration. The automatic exploration mode captures

information about interactive controls inside the windows of the application under test (AUT). On the other hand, the manual exploration mode is used to overcome situations where the automatic exploration process cannot proceed due to dependencies it cannot discover or due to password-protected functionalities. The output of the process of reverse engineering is a preliminary behavioral GUI model in Spec# together with the mapping information between the model and the implementation which is required for test execution. At a high abstraction level, the Spec# model describes the actions available to the user and their effect on the state of the graphical user interface. The mapping information consists of an XML file which stores information about the physical properties of the GUI objects and a C# code file which bridges the gap between the abstraction actions described in the model and the simulated user actions on the physical GUI objects.

Morgado et al. [Morgado11] also describes a dynamic engineering approach and the corresponding tool ReGUI, which was developed so as to minimize the effort required to extract formal and visual models for the purposes of testing. Visual models enhance a quick understanding of the functions of graphical user interfaces. The formal model is written in Spec# by the tool. At the end of execution, the ReGUI tool generates six documents: one (ReGUI Tree) which represents the ReGUI tool structure; Window Graph which represents the window map; Navigation Graph which represents the navigation map of the graphical user interface; disabled graph which represents the dependency graph; and Spec# file which is the input of the test case generation inside the AMBER iTest Project.

Mesbah et al. [Mesbah09] describe a tool, which crawls rich AJAX web applications so as to analyze and automatically reconstruct user interface states. Their study relies

on a dynamic approach, which is based on a crawler that can exercise clicks on all relevant elements in the DOM, i.e. simulate user interactions. From these state changes, a state-flow graph is constructed. It illustrates the user interface states and transitions between them. This graph can be used to generate a static mirror site which represents the style, structure and content of the AJAX application as seen in the browser sitemap which is generated after each crawling session which consists of the URLs of all generated static pages. It is indexable which enables it to show updating to any newly added state at that moment. However, it is a web-based only application, which reduces its effectiveness.

Gimblett and Thimbleby [Gimblett10] introduced a formal and generic description of UI model discovery, which is a lightweight formal method. This method constructs an interactive system model, which is discovered automatically by exploring a system's state space and simulating user actions. This is achieved through the use of standard search techniques which are augmented with domain-specific aspects such as discovery / actuation of user interface widgets. User interface components thus consist of two parts, the first part called "System Under Discovery" which contains sets of widgets, properties and values and the second part, "a state space" contains a directed graph whose nodes represent states of the device UI that is being discovered and whose edges represent user actions changing that state. This model is used as an API and as a discovery algorithm. The simulation and the discovery tool were written in Haskell and use the wxWidgets toolkit.

3.4 Summary

There have been many studies focused on reverse engineering techniques for interactive systems using both static and dynamic techniques. The aim of these studies

is to analyse the GUI systems for describing their behavior and interactive aspects. Table 3.1 shows the summary of this section, where (R-Eng) refers to reverse engineering analysis.

Authors	Type	Technique	Describe
Silva et al. [Silva11]	Static R-Eng. (using AST)	Slicing technique + formal methods	GUI Structure + interaction & state machine behaviors
Silva et al. [Silva06]	Static R-Eng. (using AST)	Using strategic programming + slicing technique	GUI Structure + behavior information
Silva et al. [Silva10]	Static R-Eng. (using AST)	GUI Surfer tool (Strategic programming and program slicing)	GUI Structure + interaction & state machine behaviors
Lin [Lin12]	Static R-Eng. (using PDG)	Formal methods (PModels + PIMs)	GUI Structure + interaction & system behaviors
Cortier et al. [Cortier07]	Static R-Eng. (using AST)	Formal validation	GUI Structure + interaction & the UI reactions in response to user actions
Staiger[Staiger07]	Static R-Eng. (source code)	Analysis GUI system implemented for the Bauhaus tool.	General program understanding
Memon et al. [Memon03]	Dynamic analysis (Without code)	(GUI Ripping technique)	GUI Structure + interaction behaviors
Paiva et al. [Paiva08]	Dynamic R-Eng.	ReGUI2FM tool	GUI Structure + interaction & system behaviors
Morgado et al. Morgado11	Dynamic R- Eng.	ReGUI tool (to reduce the effort of extracting formal models)	GUI Structure + interaction behaviors
Mesbah et al. [Mesbah09]	Dynamic R- Eng.	CRAWLJAX tool	GUI Structure + interaction behaviors
Gimblett et al. [Gimblett10]	Actual devices	Formal models (UI model discovery)	Functionality + interaction behaviors

Table 3.1. Focusing on the important aspects of some previous studies.

A number of these studies mostly covered understanding the structure and interactive behaviors of the interactive system (e.g. [Memon et al. [Memon03], Mesbah et al [Mesbah09], Morgado et al. Morgado11, Silva [Silva06], and Staiger [Staiger07]). However, there is no consideration of the underlying functionality aspects of the UIs, for our work however this is a key component, as describing both the structure and functionality behaviors of interactive systems is required to support a full system-level analysis.

Moreover, a number of these studies have applied slicing approaches to the static reverse engineering of GUIs to traverse the AST for isolating the Swing sub-program from the Java code (e.g. Silva et al. ([Silva06], [Silva10] and [Silva11])). Our approach uses the static reverse engineering analysis based on traversing the parser tree rather than using AST; we do not use slicing techniques because the final solution we propose in this work relies on the ANTLR tool that can generate the parser trees of system and their walker methods that can be used later to extract the information required. This is explained in part 4.

A similar slicing technique was used in Lin's study [Lin12] to reduce unnecessary information from program dependence graphs (PDGs) for the extraction of models. However, the drawbacks of this study are the large size of the source code in interactive systems, which produce complex dependence graphs, and thus complicate the analysis process. The other problem is coding style, where the process of capturing all information from the PDGs to build the models requires considering all possible ways of writing and designing the program. Thus, in our study, we initially have tried to simplify the PDGs by reducing the information of the large system by using the clone detection techniques. This attempt and its results are described in

section 5.1. Our study also investigates different (in both size and complexity) interactive system examples that are implemented using different coding styles to discover and address as many possible problems during the analysis process. Therefore, we have tried to use an automatic tool to generate the PDGS of a number of interactive systems to be analyzed. This approach and all its results are described in detail in section 5.2.

Other approaches used abstract interpretations to identify and capture only the GUI parts from the ASTs of the entire Java program to obtain an abstract model to capture behavioral and structural aspects of the UI system (e.g. Cortier et al. [Cortier07]). Their study creates models, which describe the event behaviors of the widgets, but without capturing these widgets and their actions, where they are described as variables that are declared in the code rather than using the widget label as represented in the UIs.

Some studies explored static analysis methods from direct source code of existing interactive applications, aimed at general program understanding (e.g. Staiger [Staiger07]), our work also has tried to analyse the direct static source code to obtain a particular set of models that represent the structural and functional behavior of GUI. This is described in detail in section 5.4.

Finally, some studies focused on describing the GUI elements and their behaviors for actual devices, (e.g Gimblett et al. [Gimblett10]) or for AJAX web applications, (e.g., Mesbah et al [Mesbah09]). Our study attempts to describe the structure and behaviors for the interactive programs in general.

Chapter IV

As we have explained, this study is based on the revers engineering of existing interactive software applications to obtain the required information for building a particular set of models, PModel (the presentation model) and PIM (the presentation interaction model). In general, reverse engineering has two methods to derive the required information, dynamic and static analysis. This project investigates and examines both techniques of reverse engineering in different experiments to try generating our required models. Thus, this work describes a number of experiments and uses several examples to examine these experiments. The next chapter, we present all of the software examples used in this study, as well as the reasons for using them. We also discuss all initial investigations and experiments performed in this paper and the problems encountered in each experiment. we then provide a summary of what we have learned from these different experiences.

Introduction to software examples

A number of software examples were used in our study, and we classify them in terms of code style (i.e., basic or enhanced examples). These programs are listed in table 4.1 along with other related information such as their code files' size, number of lines of code, number of classes. All these examples were written in the Java language using the Swing GUI library. This section ends with an outline of the reasons for choosing these examples.

Progra	Class files	Line of code	Files size
BMI calculator	2	299	139 KB
jOggPlayer	404	50403	11.5 MB
ArtOfIllusion	389	104967	23.9 MB
GoGrinder	88	13423	6.1 MB
Digital Parrot	57	7946	1.3 MB

Table 4.1. Test programs

(1) BMI calculator

The "BMI calculator" is a small interactive application to calculate body mass index (BMI) and was introduced in Section 2.2. It was created by the author in 2012, (the full source code is given in *Appendix A)*. The source code of this application is very basic and written in the simplest way - widgets are declared by using GUI Swing library class and initialized by using their constructor. Moreover, we use event listeners, which is the most common technique for handling events in Java. A listener object includes one or more event-handling methods. When an event is generated by an object, such as a button, the listener responds by running the appropriate event-handling method, which in turn contains the code that is executed when the user clicks the button. This way of writing code makes it easier to organize and understand programs. This example's screenshots are shown in Figure 2 .2 in Section 2.2.

(2) GoGrinder

GoGrinder is a game on the Android platform created by Kington in September 2012[2]. This example is an advanced open-source Swing GUI program. Its source code is well-written and documented, and it is easy to read and understand. Figure 4.1 shows the main screen and other windows of this application.

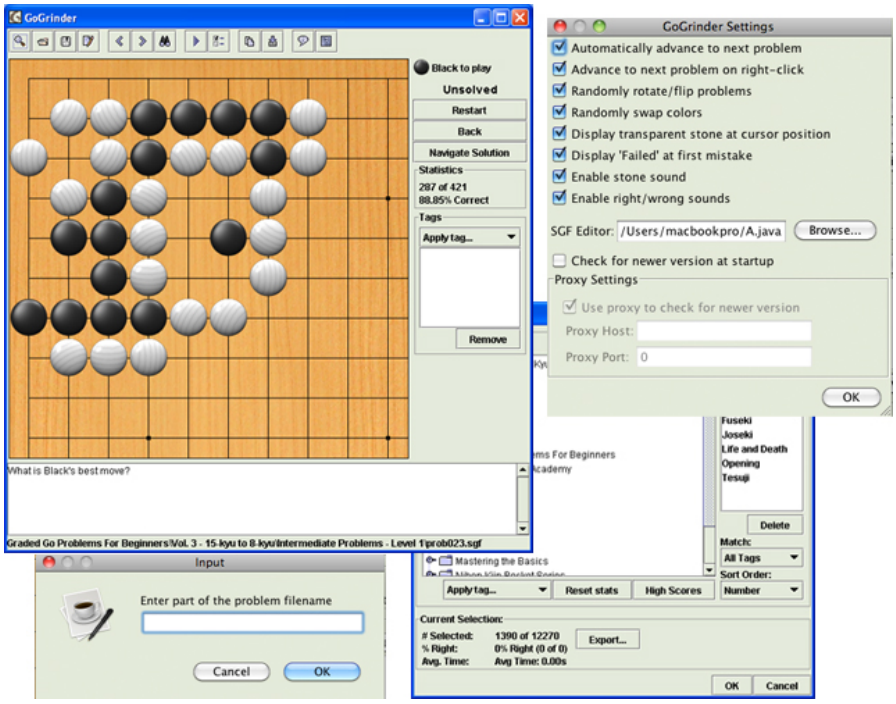


Figure 4.1. Screenshots of the "GoGrinder" app

(3) Digital Parrot

"Digital Parrot" is a software application that augments people's memory of the events of their lives. It was created by Schweer in 2011 [Schweer11]. This example is a very advanced and complex open-source Swing GUI application. The source code of the "Digital Parrot" is based on an abstract factory pattern to create widgets for a GUI environment. The factory pattern method relies on inheritance and requires the client to use the Abstract Factory interface. Different factories can be created to generate different sets of widgets that perform different actions without requiring

[2] http://gogrinder.sourceforge.net/

changes to the clients [Kuchana04]. Java software patterns are discussed in more detail in section 5.4. This example's screenshots are shown in Figure 5.2. We also use "Digital Parrot" as a complete set of models (created by hand) already exists for this software.

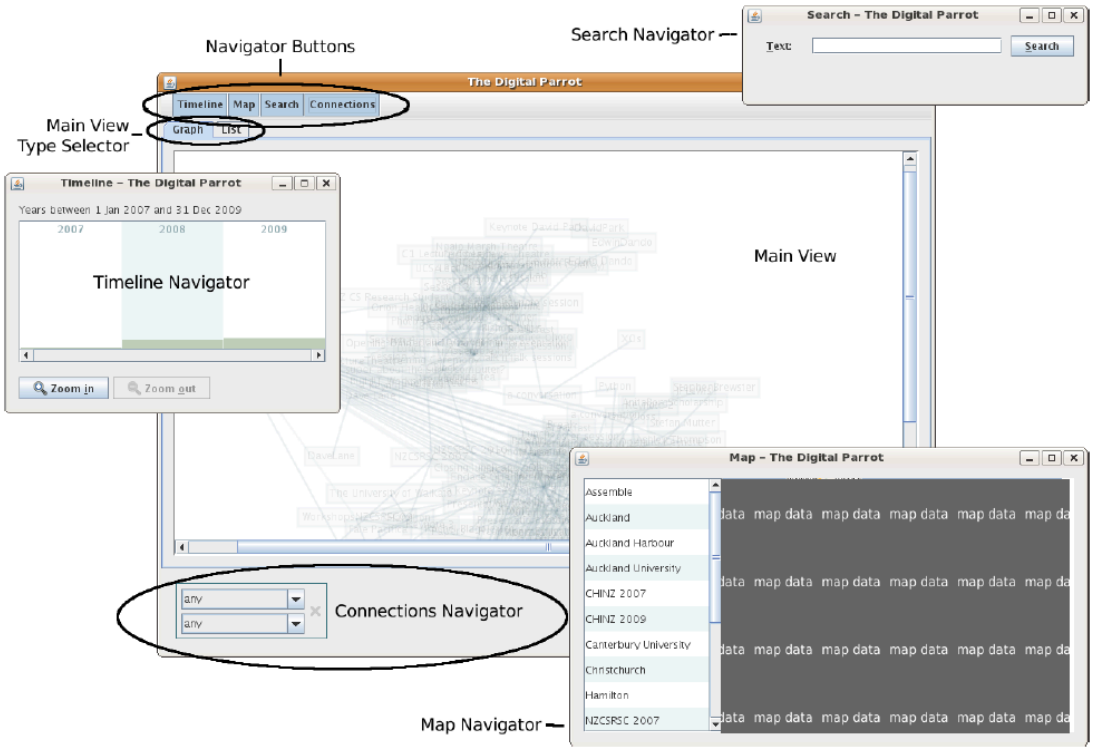


Figure 4.2. Screenshots of the main view and all navigators in the "Digital Parrot" app, (adapted from [Schweer11])

(4) Other examples

Some of the examples such as "jOggPlayer" and "ArtOfIllusion" came from 'Qualitas Corpus' web site [3], which is a curated collection of open-source Java software systems intended to be used for empirical studies of Java code. The reason for using this is because it has a number of GUI applications that are written in different code styles and programmed in advanced and complex ways; this is very helpful when testing and evaluating our different experiments during the study.

[3] http://qualitascorpus.com

Finally, we used these Java software system examples for two reasons. The first is to facilitate the analysis process in the study, because these examples represent different approaches in terms of the way the code is written. Using a gradient to classify the examples from the easiest to the most complex helped us to understand and analyse the programs. Understanding complex code can be difficult without communicating with its original programmer. Therefore, we started with the simplest examples and progressed to the most complex in order to learn the problems in every experience. For example, the "BMI Calculator" was used at the beginning, because we programmed this application and, therefore, we completely understood the code.

The second reason is to generalize the final outcome of the study. Although all the programs follow the same rules and programmatic structure to execute the code, every programmer has his/her own style of writing code. Lin's study mentioned that coding style is critical; however, it is difficult to address, and it could lead to inaccurate models of the application [Lin12]. Thus, focusing on one style is not ideal in generalizing the outcome of this study and using several different software applications with different authors ensures we address this problem.

Chapter V

Initial Experimental Approach

During this study, a number of experiments and investigations have been carried out in attempts to generate PModels and PIMs for an interactive application written in the Java programming language. Source codes were analyzed both statically and dynamically to extract all of the GUI elements to build the required models. Most of these experiments focus more on the ways to derive the PModels than PIMs, because, once PModels have been generated, the PIMs are easy to construct based on the interaction behaviors of the system that should be described with the widgets in the PModels itself. This section explores in detail all of the initial experiments performed in this study, and the problems encountered in each experiment. The chapter ends by giving a summary of all these conducted experiments.

5.1 Experiment 1: Clone detection using program dependence graphs

Introduction

Duplicate code is separate fragments within an application's code that are very similar or identical. Code clones are a common phenomenon in any software system, that might be the result of the process of copying and pasting or produced accidentally e.g. when using APIs and libraries. In section 2.3, we introduced code clones and explained their relevance to revers engineering.

Lin study [Lin12] showed that program-slicing techniques can be combined with a program dependency graph (PDG) of the system, which helps to create PModels and PIMs. The general process of reverse engineering an interactive system suggested by Lin is shown in Figure 5.1.

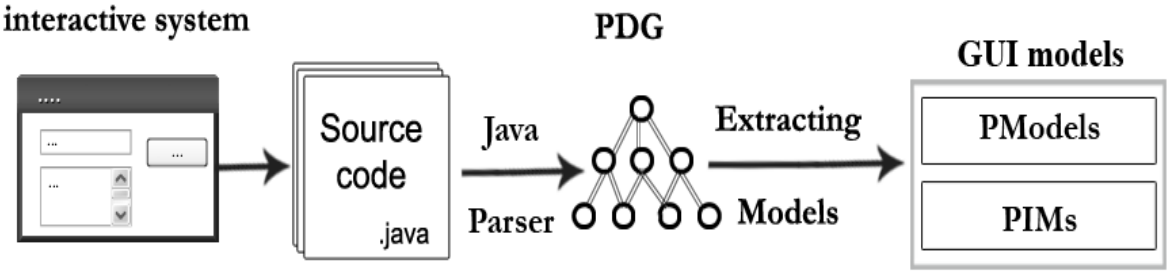


Figure 5.1. General process in [Lin12] of reverse engineering an interactive system

Based on some of the problems identified by Lin (such as the size of PDGs), clone detection might help to reduce the complexity of source code; whether it could support the interactive reverse engineering process was also investigated. This is clarified in Figure 5.2. This section introduces all the tools used in this experiment, as well as the three different methods used to reduce and simplify the code information and the PDGs based on the clone code outputs. The problems encountered and the results obtained with each method are described in detail.

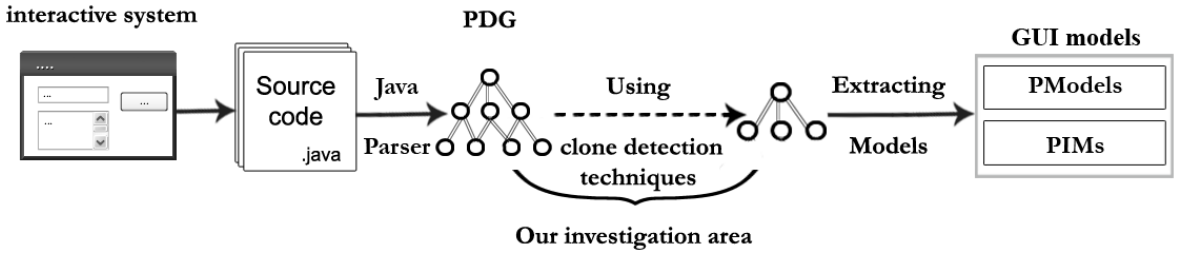


Figure 5.2. The investigation area, which aims to simplify and reduce the complexity in the analysis process

Tools

Initially, Dependency Finder[4] was used to extract the PDGs and to mine them for useful information. This tool is freely available and is useful for understanding the structural complexity of code, because it shows the dependencies between high-level components.

The Dependency Finder tool constructs dependency graphs based on the information in class files of the program. Generally, dependencies occur when a component uses the services of another component. This can happen, for instance, when a class inherits from another, has an attribute whose type is of another class, or when one of its methods invokes a method or field access on an object of another class. Thus, a dependency is when the functioning of one component ***A*** requires the presence of another component ***B***. That is ***A*** depends on ***B***, and it can be said that ***A*** is a dependent and ***B*** is dependable, and this can be as follows:

$$A \rightarrow B$$

That is ***A*** has an outbound dependency while ***B*** has an inbound dependency, and can be graphed as shown below in Figure 5.3.

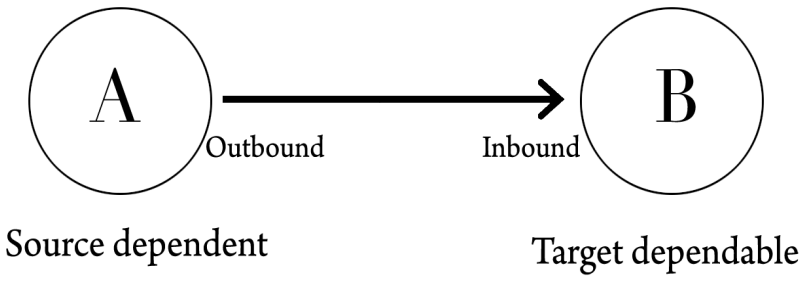


Figure 5.3. Dependency relationships.

[4] http://depfind.sourceforge.net/

Using this tool, PDGs are generated as text in XML files where <inbound> and <outbound> tags are used to present the dependencies of the Java programs. An example of part of the PDG for the "BMI calculator" application is presented in Figure 5.4.

```
- <dependencies>
  - <package confirmed="yes">
      <name>bmi_ex</name>
    - <class confirmed="yes">
        <name>bmi_ex.BMI_ex</name>
        <outbound type="class" confirmed="no">java.lang.Object</outbound>
      - <feature confirmed="yes">
          <name>bmi_ex.BMI_ex.BMI_ex()</name>
          <outbound type="feature" confirmed="no">java.lang.Object.Object()</outbound>
        </feature>
      - <feature confirmed="yes">
          <name>bmi_ex.BMI_ex.main(java.lang.String[])</name>
          <outbound type="feature" confirmed="yes">bmi_ex.MBI_calculater.MBI_calculater()</outbound>
        - <outbound type="feature" confirmed="no">
            bmi_ex.MBI_calculater.setDefaultCloseOperation(int)
          </outbound>
          <outbound type="feature" confirmed="no">bmi_ex.MBI_calculater.setLocation(int, int)</outbound>
        - <outbound type="feature" confirmed="no">
            bmi_ex.MBI_calculater.setLocationRelativeTo(java.awt.Component)
          </outbound>
          <outbound type="feature" confirmed="no">bmi_ex.MBI_calculater.setSize(int, int)</outbound>
          <outbound type="feature" confirmed="no">bmi_ex.MBI_calculater.setTitle(java.lang.String)</outbound>
          <outbound type="feature" confirmed="no">bmi_ex.MBI_calculater.setVisible(boolean)</outbound>
          <outbound type="class" confirmed="no">java.awt.Component</outbound>
          <outbound type="class" confirmed="no">java.io.PrintStream</outbound>
          <outbound type="feature" confirmed="no">java.io.PrintStream.println(java.lang.String)</outbound>
          <outbound type="class" confirmed="no">java.lang.String</outbound>
          <outbound type="feature" confirmed="no">java.lang.System.out</outbound>
        </feature>
      </class>
    - <class confirmed="yes">
        <name>bmi_ex.MBI_calculater</name>
        <outbound type="class" confirmed="no">javax.swing.JFrame</outbound>
```

Figure 5.4. Example of part of the PDG generated for a "BMI calculator" program, which was produced using the Dependency Finder tool.

To produce a visual representation of the graph from the XML requires the transfer of the resulting XML either manually or by using a specialist tool. This example used Adobe Photoshop CS3 to manually draw the PDGs generated for "BMI Calculator"; this is shown in Figure 5.5.

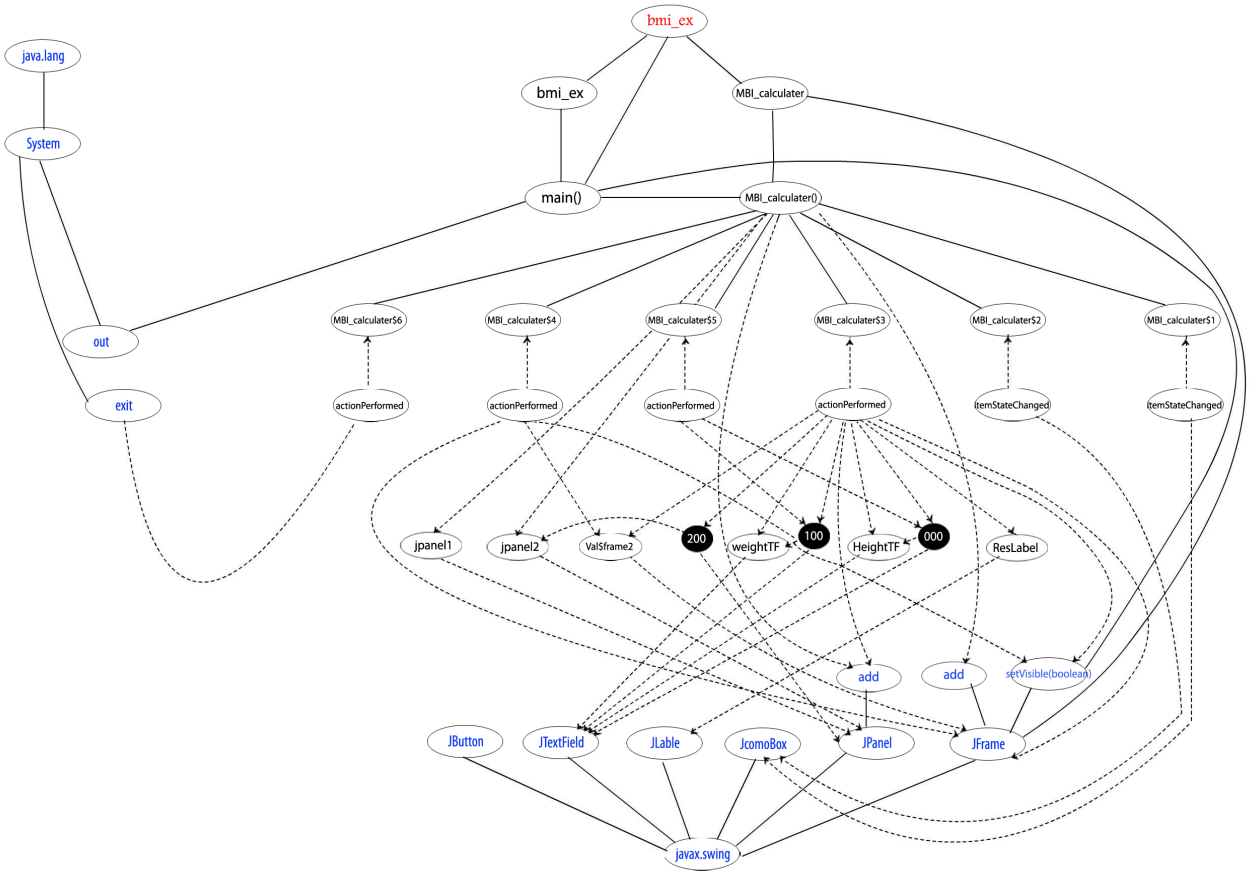


Figure 5.5. PDG generated for the "BMI Calculator" program.

The CCFinderX tool [5] was used in this experiment to detect duplicate code in programs. This tool is a token-based code clone detector, which detects duplicated code fragments in source files written in several programming languages (Java, C/C++, COBOL, VB, and C#). The CCFinderX tool was used because it is freely available, it works with Java code, and it provides complementary matching techniques. Examples of the CCFinderX output produced for the "BMI calculator" application and the clones detected are shown in Table 5.1.

[5] http://www.ccfinder.net

Set	Fragments 1	Fragments 2

1	`weightTF=new JTextField();`
	`weightTF.setBounds(140,40,60,20);`
	`jPanel1.add(weightTF);`
		`HeightTF=new JTextField();`
		`HeightTF.setBounds(140,90,60,20);`
		`jPanel1.add(HeightTF);`

2	`final String W[]={"kilograms","pounds"};`
	
	`final JComboBox WCB=new JComboBox(W);`
	`WCB.setBounds(220,40,125,20);`
	`jPanel1.add(WCB);`
	`WCB.addItemListener(new ItemListener(){`
	`public void itemStateChanged(ItemEvent ie){`
	`str = (String)WCB.getSelectedItem();`
	`}`
	`});`
		`final String H[]={"inches", "centimeters"};`
		`final JComboBox HCB=new JComboBox(H);`
		`HCB.setBounds(220,90,125,20);`
		`jPanel1.add(HCB);`
		`HCB.addItemListener(new ItemListener(){`
		`public void itemStateChanged(ItemEvent iee){`
		`str2 = (String)HCB.getSelectedItem();`
		`}`
		`});`

3	`public void actionPerformed(ActionEvent event)`
	`{`
	`frame2.setVisible(false);`
	`}`
		`public void actionPerformed(ActionEvent event)`
		`{`
		`System.exit(0);`
		`}`

Table 5.1. Three clone sets found by the CCFinderX tool, where each set contains two fragments of clones.

Experimental procedure

The clone detection technique was used to reduce the information in the source code, which may help to reduce and simplify the complexity of the PDGs, as well as facilitating the analysis process to generate the PModels and PIMs. The CCFinderX tool was used to detect duplicate code in some of the example applications described previously in this chapter, and the PDGs were examined and analyzed based on the results for the clone sets to derive the models.

To identify the clone areas in the PDGs, all of the nodes in the "BMI Calculator" PDG were colored. This represented the clone sets for the code, where each clone pair was the same color. There were three clone sets in this example, as shown in Table

5.1, and each of these sets was highlighted using the same color to distinguish them. In this case, red, green, and blue colors were used (see Figure 5.6).

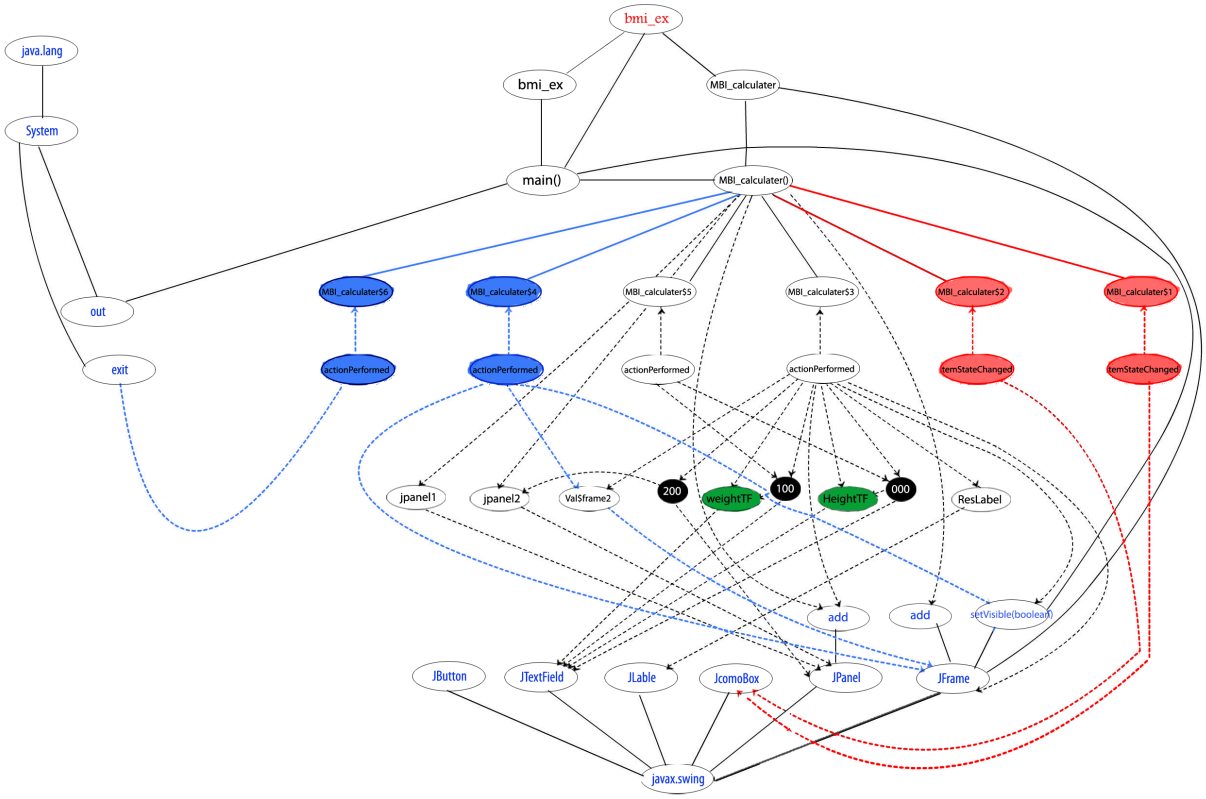


Figure 5.6. Highlighting of the clone sets for the “BMI Calculator” PDG.

The important question is: “How can we extract the required models based on the clone set outputs?” Three different methods were used to answer this question. These methods were used to examine the effects of applying clone detection techniques to the code and to the overall structures of PDGs, in terms of reducing the information. This could then reduce the complexity of the analysis process required to extract the models. Moreover, an attempt was made to extract models of the “BMI calculator” example based only on its code, rather than the analysis of its PDGs, because of a deep understanding of this example code. A complete description of the analysis and the extraction of the models of the PDGs can be found in Section 5.2.

Method 1

This method for extraction of the PModels from the clone fragments only was attempted, as it was less complex, as well as easier to understand and analyze. This seems like a sensible approach because of the knowledge that using widgets from the Java Swing library leads to an abundance of cloned code due to the repetition of code patterns, so it is likely that the clone fragments will represent widgets. For the "BMI Calculator" application examples, the extracted PModels based on the clone sets were approximately as follows.

```
(WeightEntry, Entry, ())
(WeightSel, Container, ())
    (KilogramsItem, SvalSelector,())
    (PoundsItem, SvalSelector,())
(HeightEntry, Entry, ())
(HeightSel, Container, ())
    (inches, SvalSelector,())
    (centimeters, SvalSelector,())
(QuitButton, ActionControl, ((QuitApp))
(CloseButton, ActionControl, (I_Close))
```

Based on the models above, it was found that the full PModels could not be created from these duplicate fragments (unlike the models described in section 2.2). This is because this method ignored some elements that were important for creating models, such as the "Calculate" and "Clear" buttons, and all of the frames etc., which were not present among the clone fragments. Thus, this method was not successful at extracting all of the information required to produce the models, which meant that it could not be used for the PDG analysis process.

Method 2

In this method, the opposite of the previous method was attempted. It was investigated whether this approach could help to reduce the complexity of the code and its PDGs using the clone technique. The aim was to inspect all of the code fragments in an interactive application file and to keep only one copy of each set,

while removing all of the other duplicated codes. For example, if a particular set contained four duplicate code fragments, only one copy was kept and the other three code fragments were removed. Table 5.2 shows a comparison of the state of each program before and after deleting the duplicate fragments, in terms of the number of lines of code and the nodes in the PDGs.

Program	Original code information			Code information after delete copies in the clone sets, except one copy	
	Lines of code	Nodes in PDG	Clone sets	Lines of code	Nodes in PDG
BMI calculator	299	30	3	244	28
jOgg Player	50403	5256	200	38719	4793
Art Of Illusion	104967	9805	5000	72782	8499

Table 5.2. Comparison of the differences in each program before and after removing the duplicate clone fragments.

From Table 5.2, it is clear that there was a reduction in the number of lines of code and nodes in the PDGs of all the programs after using this algorithm, which retained one copy of each set and removed the other copies. Figure 5.7 shows the PDG graph for the "BMI Calculator" application using this method, where all the red parts represent the clone code fragments deleted from the graph. There is clearly a relative change in the graph, because there is less information than the original graph, which will probably reduce the effort required to analyze the application. Using this algorithm, the PModels extracted for the "BMI Calculator" program were as follows.

```
MainWin is
  (WeightSel, Container, ())
     (KilogramsItem, SvalSelector,())
     (PoundsItem, SvalSelector,())
  (CaculateButton, ActionControl, (S_result, I_Calculate))
  (ClearButton, ActionControl, (S_Cleare))
  (QuitButton, ActionControl, ((QuitApp))
ResultWin is
  (Result, SvalueResponder, (S_result)
```

Unfortunately, the model extracted did not match the correct model for the "BMI calculator" (as described in section 2.2). Thus, we found that this method reduced the information in the code, which may help the analysis, but like the previous method, it failed to create the full PModel because some GUI widgets were missing due to the deletion process. To address this problem, we had to modify this algorithm to make it more effective, by reducing the amount of information in the code while retaining the basic information (such as widgets, their behaviors, and related information), which were important for extracting the models. The new adjustment algorithm is described in the next method.

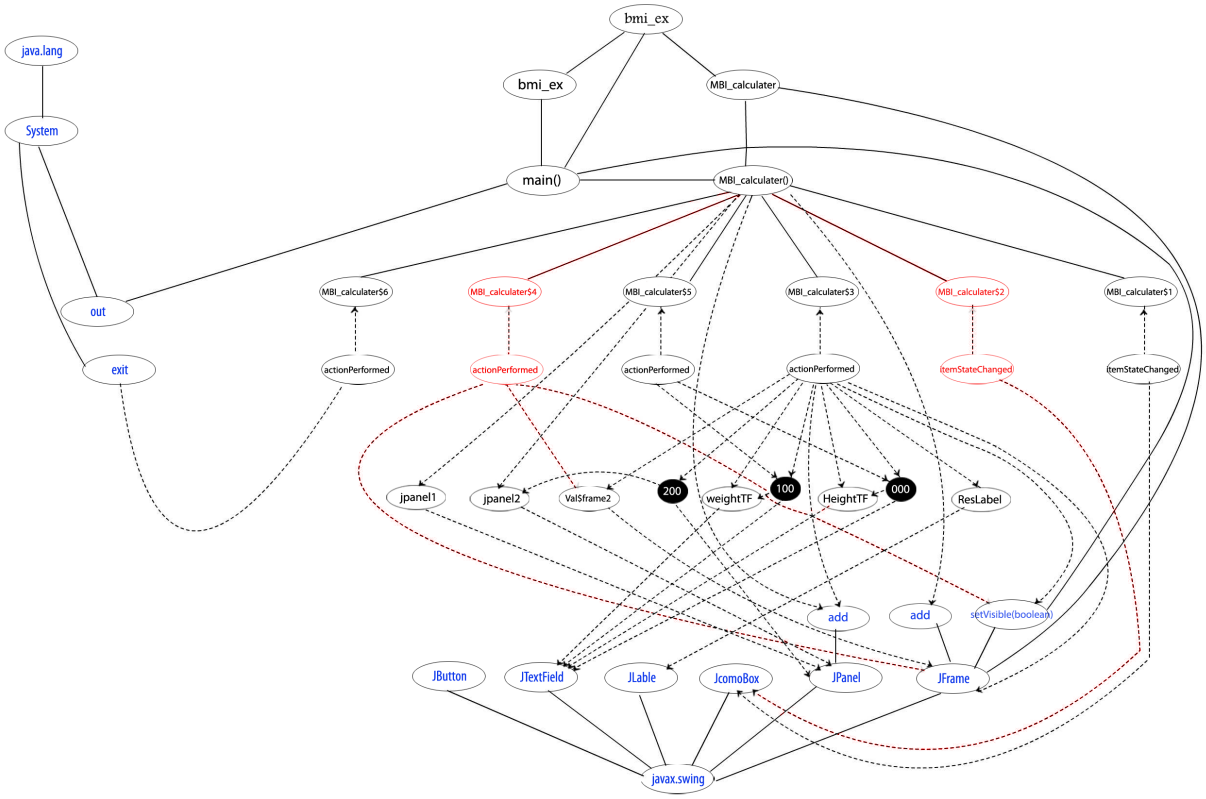


Figure 5.7. The PDGs obtained after deleting the duplicate clone fragments for the "BMI Calculator" program.

Method 3

As explained previously, Method 2 was an effective algorithm for reducing the size and quantity of code; because it removed all of the clone fragments from each clone set, with the exception of one copy. However, much of the important information required to build the models was also deleted. To avoid this problem, the steps used in Method 2 were followed, but a specific heuristic procedure was used that helped to reduce the information in the code while maintaining the basic information. Basic information was defined as that required to extract the models, such as widgets, initializers of widgets, declarations, and call function statements. A widget could be extracted from the data types of codes, while a widget's name could be determined from the widget's initializer statement in a GUI system (which represents a constructor of the widget). Function declaration statements are also very important because they help to identify the behaviors of widgets, where a function can be the event handler for a widget. Finally, statements that refer to function calls may help to identify behaviors and their types, i.e., systems or interactions, such as the "SetVisible (true)" method, which is responsible for opening or displaying windows.

Thus, the heuristic algorithm comprised the following steps:

Step 1: Keep one copy from each clone set.

Step 2: Read each statement in the other clone fragment copies and remove them if they do not include one of the following statements:

- Widgets and function declarations;
- Initializer of widget statements;
- Function calls statements.

This algorithm was applied to a number of interactive application examples. Table 5.3 shows the changes that occurred in each program in terms of the number of lines of code and the nodes in the PDGs.

Program	Original code information			Code information after using the heuristic method	
	Lines of code	Nodes in PDG	Clone sets	Lines of code	Nodes in PDG
BMI calculator	299	30	3	254	29
jOgg Player	50403	5256	200	40326	4972
Art of Illusion	104967	9805	5000	80515	9149

Table 5.3. Information related to the number of lines of code and the nodes in the PDGs for three examples using Method 3.

The extracted PModels for the "BMI Calculator" program using this method were as follows:

```
MainWin is
   (WeightEntry, Entry, ())
   (WeightEntry, SvalueResponder, (S_clear))
   (WeightSel, Container, ())
     (KilogramsItem, SvalSelector,())
     (PoundsItem, SvalSelector,())
   (HeightEntry, Entry, ())
   (HeightEntry, SvalueResponder, (S_clear))
   (HeightSel, Container,())
         (CentimeteressItem, SvalSelector,())
         (InchesItem, SvalSelector,())
   (CaculateButton, ActionControl, (S_result, I_Calculate))
   (ClearButton, ActionControl, (S_Cleare))
   (QuitButton, ActionControl, ((QuitApp))
ResultWin is
   (Result, SvalueResponder, (S_result)
   (CloseButton, ActionControl, (I_Close))
```

The models above are correct and match to the target models. Thus, using the heuristic in this experiment is very helpful to produce the correct and full models. But, the main question is: Does the use of code duplicate detection techniques in this experiment achieve the goal of this study in terms of reducing the complexity of the

source code to facilitate the analysis process for extracting the required models from the PDGs? To answer this question, it is necessary to compare the results of the heuristic experiment with the original information, in terms of the number of lines of code and the nodes in the PDGs. Figure 5.8 shows the overall comparisons based on the lines of code numbers, and nodes on the PDGs in the three examples.

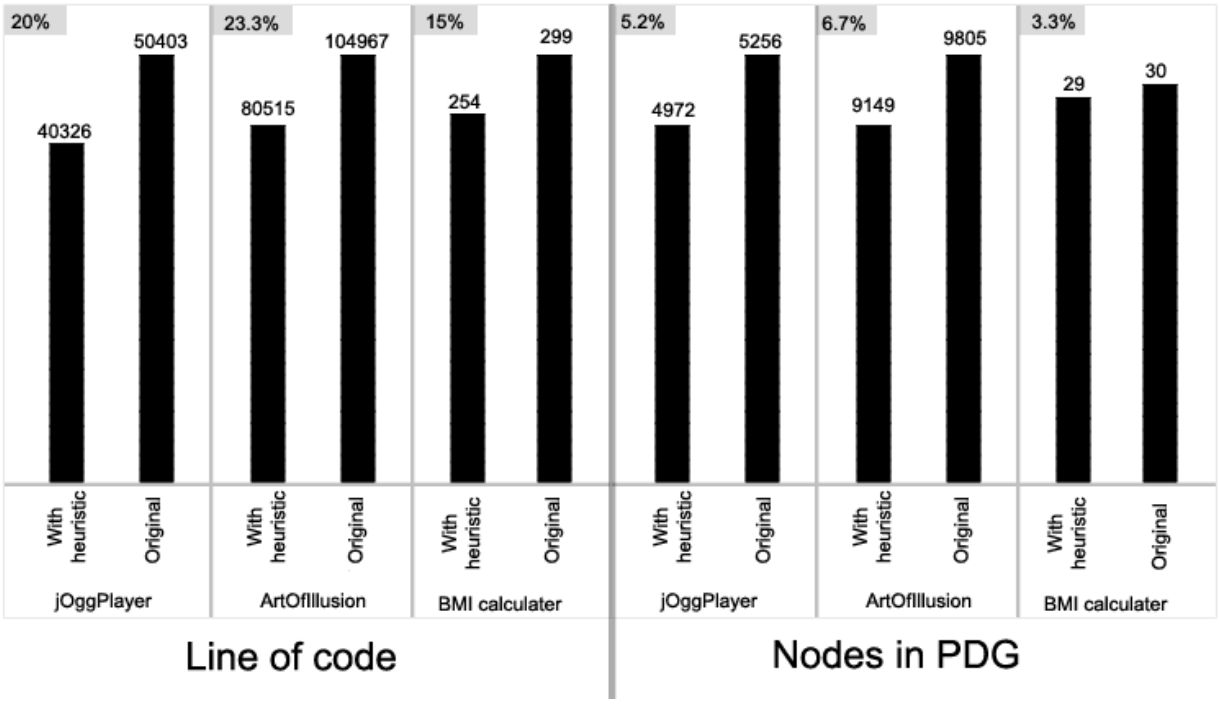


Figure 5.8. Comparison of the number of the lines of code and the nodes in PDGs for the three examples.

As shown in Figure 5. 8 above, the number of lines of code in the heuristic experiment was reduced in all three examples compared to the originals; the number of lines of code is decreased by 15 per cent in "BMI Calculator", 23.3 per cent in "Artofillusion", and 20 per cent in "JoggPlayer". Also, there were only slight changes in the number of nodes in the PDGs. The proportion of the decrease in nodes in the PDGs was 3.3 per cent in "BMI Calculator", 6.7 per cent in "Artofillusion", and 5.2 per cent in "JoggPlayer".

Although these results show that the clone detection technique along with using the heuristic to reduce the information was reducing the amount of information in the source code and PDGs, it does not support the analysis of PDGs in an interactive system. This is because the decrease is very slight and this cannot help in reducing the complexity of the PDGs during the analysis process. The amount of work required (computationally) to identify the clones and apply the heuristic is greater than savings produced by the reduction in size of the PDG.

5.2 Experiment 2: Extracting models from PDGs

As explained previously, a PModel represents a window or frame in a GUI, and all the widgets inside this window are represented by the triples in the PModel. Each widget in the PModel is described in terms of three components: a name, a category, and a set of behaviors. To get the final PModel, the widgets need to be determined and then these three components from each widget defined in the PDGs. Although a detailed explanation for extracting PModels and PIMs from PDGs is presented in [Lin12], this study also explains this extraction in detail. The major difference is that in this research we use an automatic tool, 'Dependency Finder', to generate PDGs, whereas Lin's study constructed the PDGs for interactive software applications manually after generating the AST from the JavaParser tool. There are two reasons that this study does not use the 'JavaParser' tool to produce the PDGs. Firstly, in this study, a number of examples are investigated, so using an automatic tool to generate the PDGs helps to reduce the effort and time required. Secondly, this tool works only with Java 1.5, and there is no update release to support other Java programs versions, such as 1.6 or 1.7, that have some syntax changes. This section explains in detail the method used to extract the widgets and their behaviors from the PDGs and presents it as a further step in advancing the automation of this process.

Detecting the widget

Graphical user interfaces contain a number of widgets such as buttons, checkboxes, menus, etc. which are responsible for capturing interaction from the user. These widgets can be defined by their data types. The widget type can be used as part of the name in the model, if the type of the widget in the PDGs is known. The name of these widgets is taken from the widget's label and used to identify it in the PModel. The label describes the widget in the GUI. For example, if a button is labeled *Cancel* in a

GUI, the matching widget in the PModel could be named ***Cancel_button*** or ***Cancel_btn***, and if a *JTextField* is used to enter data such as nationality; the widget in the model could be ***nationalityEntry.***

Figure 5.9, shows an extract taken from the full PDG for the "BMI calculator" program (see Figure 5.5) to simplify the analysis process.

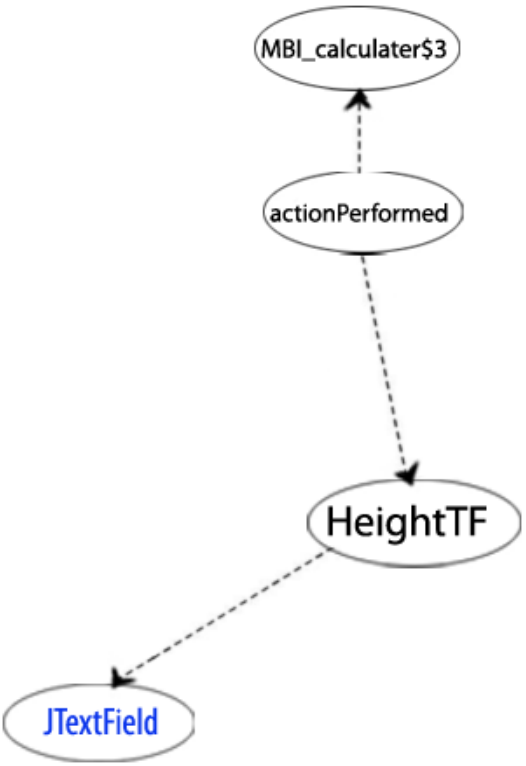


Figure 5.9. A part of PDGs for the "BMI calculator" program

There is a ***HeightTF*** node, which is dependent on the *JTextField.* This means ***HeightTF*** is a widget variable and it is declared as type *JTextField*. It is clear that no information is available about this widget's label in the graph. Thus, the widget variable and the widget's type, separated by a "–", will be used to describe the name in the PModel as follows:

(Height–TFEntry, …….

Moreover, if there is a widget, such as a frame, its type is available in the PDG but without any information about its title or label. In this case, ***Wind*** is given as a default name. So, when referring back to the complete PDG in Figure 5.5, the widgets and their types can be identified from the PDG based on this technique. Table 5.4 shows the widgets extracted in this way.

Dependency nodes	Widget types	Default name
bmi_ex.MBI_calculater nod → ***JFrame*** node	JFrame	Wind_1
MBI_calculater() node → ***JFrame*** node	JFrame	Wind_2
HeightTF node → ***JTextField*** node	JTextField	HeightTF
weightTF node → ***JTextField*** node	JTextField	weightTF

Table 5.4. Extracted widgets based on their types

Table 5.4 shows all the widgets identified based on their types. However, for some widgets, no information is available about their labels or types in PDGs. Instead, they are identified by their event handler methods, such as the ***actionPerformed*** node or the ***itemStateChanged*** node. Programmers usually use the ***actionPerformed*** method to create an event for widgets, such as buttons, radio buttons, and checkboxes, whereas they use the ***itemStateChanged*** method to respond to one element in a list of items in a ComboBox widget.

In this case, a widget can be identified by an "event handler method" node, where the method is responsible for any event that occurs for a certain widget. Figure 5.9 shows that the ***bmi_ex.BMI_calculator$3*** node is dependent on the ***actionPerformed*** node. The ***actionPerformed*** method is used by various widgets, such as the button, radio button, and checkbox widgets. This means there is a widget that uses this method, but no node is mentioned with its type or its widget label in this PDG. Therefore, based

on the relevant method (e.g. ***actionPerformed)***, it can be inferred that there is a widget, and then "*Widget*" will be assigned as the name of this anonymous widget to be represented in the PModel, as follows:

(Widget, …….

Anonymous widgets mean that the widget was detected by its event handler method and it is hard to identify their type. The following table shows the method nodes and the widgets identified from these that can be extracted from the full PDG shown in Figure 5.5.

Expected widget	Dependency nodes
ComboBox	***bmi_ex.MBI_calculater$1*** node → ***itemStateChanged*** node
	bmi_ex.MBI_calculater$2 node → ***itemStateChanged*** node
Button, Radio button, or Checkbox	***bmi_ex.MBI_calculater$3*** node → ***actionPerformed*** node
	bmi_ex.MBI_calculater$4 node → ***actionPerformed*** node
	bmi_ex.MBI_calculater$5 node → ***actionPerformed*** node

Table 5.5. Shows the expected widget types based on their event handler methods

Detecting the widget category

Up to this point of the analysis, all the widgets can be extracted from the PDGs, based on identifying the data type that belongs to GUIs or identifying the method that is used by the widget to create events. Once the widget is found, determining the category of the widgets is required to build the PModels. According to Lin the category of a widget in a PModel can be defined by their type [Lin12]. In this study, the category can be obtained in two ways. Firstly, similarly to Lin's study, widgets can be categorized by their type, e.g., button, combo box, or text field. For example, the category of a text field ***HeightTF*** is ***Entry***, and then the PModel is as follows:

(HeightTFEntry, Entry, …….)

Secondly, identification of the category of the widgets that do not mention their types in PDG is by identifying their "event handler method" node, such as ***actionPerformed***, ***itemStateChanged***, etc. Through these functions, a node can be used to determine the categories of widgets. Referring again to the Figure 5.9, the widget node represents an ***actionPerformed*** method that is used by button, radio button, and checkbox widgets. All three widgets types are categorized as ***actionControl*** in the models, as follows:

(Widget, actionControl, ……..

This can be generalized further; any widget that has an ***actionPerformed*** node can be categorized as an ***actionControl*** irrespective of its actual implemented type.

Detect widget behaviors

Lin's study stated that behaviors can be extracted by navigating the procedure calls for the widgets that have event-based controls [Lin12]. Based on this study, the widgets' behaviors were identified by traversing the procedure calls for these widgets to determine the behavior, either S_behavior, I_behavior, both or none. For further clarification, how to identify the different types of behaviors (I_behavior and S_behavior) in the PDGs is explained.

Interaction behavior (I_behavior)

I_behaviors represent the navigation between the windows in the application. These windows include all the frames and dialog boxes in the application. To identify the I_behaviors, a focus is needed on all methods that are responsible for creating the windows and their actions in the code of the Java application. A window can be created by using window types in the Swing library, such as ***JFrame*** and ***JWindow etc.*** Special methods can define the window's interactions; for example, the ***setVisible***

method is used to show or hide the window. Table 5.6 shows a list of widget types and their methods that are considered as interaction behaviors.

Widget Type	Method Name	Description
JFrame, JDialog, JWindow, OptionPane	setVisible(boolean)	Shows or hides the component depending on the value of the Boolean parameter **Parameters**: boolean: If **true**, shows this component; otherwise, hides this component
JOptionPane	showConfirmDialog	Displays a dialog box to ask the user to confirm by requiring an answer, such as **yes**, **no**, or **cancel**
JOptionPane	showInputDialog	Prompts for an input
JOptionPane	showMessageDialog	Notifies the user about something that has happened
JOptionPane	showOptionDialog	Displays a dialog box that combines the three methods ***showConfirmDialog***, ***showInputDialog***, and ***showMessageDialog***
JFileChooser	OPEN_DIALOG, SAVE_DIALOG, etc.	Provides a simple mechanism for the user to choose a file

Table 5.6. Examples of methods [6] that can help to identify the I_behaviors.

In PDGs, the I_behaviors can be obtained by tracking all nodes that connect with the widget node and by checking whether each connecting node calls one or more of the methods listed in Table 5.6. This is partially dependent on programming style: some programmers prefer to create a separate function, which calls other functions; for example,

> ***actionperformed*** node --> ***Close*** ( ) --> ***setVisible (Boolean***)
>
> Where, ***Close ()*** is a function created by the programmer and this function in turn calls the ***setVisible (Boolean)*** method.

[6] http://docs.oracle.com/

In this case, the set of all nodes that can be reached by following downstream or upstream dependencies from a starting node needs to be tracked, and checked for whether any of these nodes calls one of the methods listed in Table 5.6 in order to determine if the call is an I_behavior or not.

System behavior (S_behavior)

An S_behavior is any behavior that affects the underlying functionality of the system. This behavior is identified from PDGs in two ways: (1) by determining the connection between the widget node and a field access node; and (2) by traversing the procedure calls for the widgets.

The first way is by checking the dependency between the widget node and any field access node. The field access node is a container used by system or program processes either to store the value of a certain variable or to track the temporary values of a variable before returning the final value. This behavior is an S_behavior, because it affects the underlying functionality of the system. Thus, when there are dependencies between a widget node and the field access node, the widget's behavior is an S_behavior.

The second way is by tracking the procedure calls from the widgets. If the node connects with one of the methods that affect the interaction window, the behavior is an I_behavior. If this node also connects with other procedure nodes, then the widget has multiple behaviors, which may be both S_behaviors and I_behaviors. If the widget node calls methods that do not use any of the methods listed in Table 5.6, then the behavior is an S_behavior. If the widget does not connect with the field access node or any procedure node, the widget has no behavior.

Finally, if the widget quits the application by connecting with an ***exit*** method node (from the Swing library under the System Class), then ***QuitApp*** is used as the behavior, as follows:

(Widget, actionControl, (QuitApp))

Examples

As explained earlier, the set of all behaviors can be extracted based on the dependencies of the widget node with other nodes that call procedures or connect with field access nodes. Using the "BMI Calculator" example for clarification, and keeping the example very simple to understand the analysis process, Figure 5.10 illustrates part of the PDG for the "BMI calculator" program (Figure 5.5, in section 5.1 for full PDG).

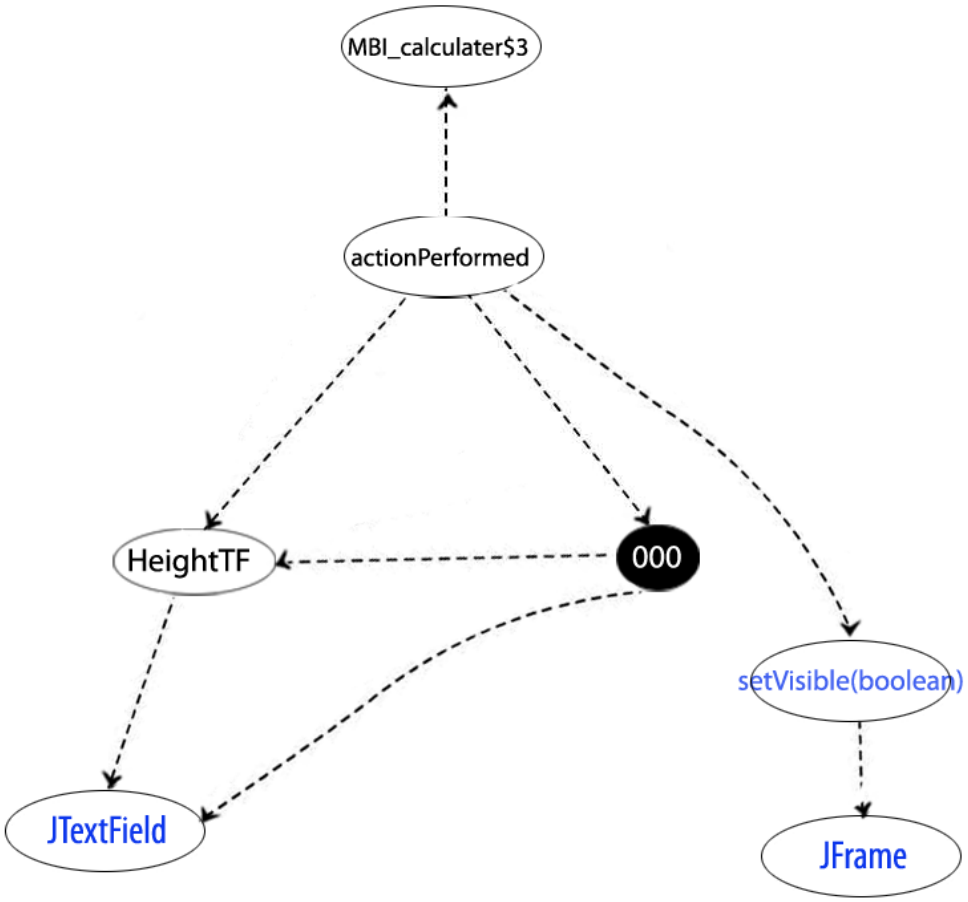


Figure 5.10. Parts of "BMI calculator" program's PDG

The ***BMI_calculator$3.actionPerformed*** node connects to the access field node; ***access$000***, which indicates an S_behavior. This widget also exhibits an I_behavior, because the ***setVisible (boolean)*** node depends on this node, which is one of the methods listed in Table 5.6. Therefore, the final form of the PModel of this widget is:

(Widget, actionControl,(I_behavior ,S_behavior))

Additionally, the field access ***access$000*** in the Figure connects with two ***actionPerformed*** nodes, where it sends the data to the user interface through a system process or it receives input data from the user to be used by the system. The access name ***$000*** depends on the ***HeightTF*** variable, which is of type ***TextField*****;** Therefore, any node that connects to this field will require the ***HeightTF*** widget.

To extract the PModel from the PDG for this widget node, the purpose of the ***TextField*** variable used is required. ***TextField*** is commonly used to allow a user to input data needed by the program, and the widget is typically used only for data entry and does not have any behavior. Therefore, it is described in PModels as:

(HeightTF_ entry, Entry, ())

TextField can also be used to get the results of a system process; e.g., when a user presses a certain button, the ***TextField*** displays the result. Thus, this behavior is an S_behavior, and the widget is described in the PModels as:

(HeightTF_entry, svalueResponder,(S_behavior))

Thus, we can derive all the required information to extract the PModels from the PDGs. For further clarification, the next section gives a summary of the algorithm above.

Summary

This section shows how the PDGs of an interactive application to create PModels are analyzed. The first step in this analysis is to detect the widgets, which can be done in two ways: widget's data type (its name in the code will also be used as the name of this widget in the model, and its category can be identified based on this type); and widget's method for those that do not mention its data type in the graph (in this case it is called an anonymous widget and its name written in the model as 'Widget' and its category will be based on the method).

Detecting the category in PModels of the widget is explained in this section. The category of the widget can be determined based on the widget type, e.g., button, combo box, or text field. In addition, there are some widgets in the graph that do not mention their types and which are identified by their related methods, that is responsible for the events for these widgets. By identifying such methods and their dependencies with the widget, widgets in the PModels can be categorized.

Finally, this section explores how all behaviors of each widget can be detected from the PDGs. Widget behaviors can be identified by traversing the procedure calls for the widget and then determining the behavior of the widget, either system or interaction behavior, both or none. Finding any widget node connected with a field access node can help to identify the S_behavior type.

Implementation

By following the algorithms described in the previous section, a tool was created to automatically analyze the PDGs and extract all GUI elements to create the PModels. This system was implemented using Java and the NetBeans IDE. This tool can read and analyze all the information in PDGs. However, it cannot generate correct and full PModels due to the absence of some required information in the PDGs. This section explores the steps followed to analyze PDGS and implement the analyzer tool in this experiment and discusses the problems that limit the building of a complete model from the PDGs.

In this experiment, the Dependency Finder[7] tool is used to generate the PDGs automatically and export them into an XML file. Our tool, PDGs' analyzer tool then reads the XML document by using SAX Parser, which is the most commonly, used XML parser in Java. SAX parser uses three callback methods to parse and read the XML: ***startElement()*** method, which is used to get a opening tag '<'; ***endElement()***, which is used to get a closing tag '>'; and ***character()***, which is used to get a simple character string.

Result and problems identified

The PDGs analyzer tool has been implemented and the SAX parser is used in the tool to facilitate the process to read the XMLfile and extract the required information from the PDGs based on the algorithm described in this section. This tool was run with the "BMI calculator" example and also on the other applications described in Chapter 4. Figure 5.11 presents an example of the output PModels for the "BMI calculator" application.

[7] http://depfind.sourceforge.net/

```
The final PModels obtained from PDGs are :

Wind_1 is …..

Wind_2 is ……

(HeightTF, Entry,())

(HeightTF, svalueResponder,(S_behavior))

(weightTF, Entry,())

(weightTF, svalueResponder,(S_behavior))

(ComboBox, Selctor,())

(ComboBox, Selctor,())

(Widget, actionControl,(I_behavior,S_behavior))

(Widget, actionControl,(I_behavior))

(Widget, actionControl,(S_behavior))

(Widget, actionControl,QuitApp)
```

Figure 5.11. The PModels for "BMI calculator" app extracted by using the PDGs' analyzer tool.

The tool extracted all the widgets and the types of their behaviors. Unfortunately, the tool cannot generate the correct and full model of this interactive example. It can be seen that the name of each widget as shown in the GUI is hard to determine and all of the widgets that are identified based on their methods have the same name "***widget"***, which makes it difficult to distinguish between them. Also, their behavior types are extracted but without identifying the names of these behaviors, which is important to describe in the PModels. Moreover, the widgets are represented separately and it is difficult to recognize whether these widgets belong to any frame or window in the

PModel; this does not help to capture the appearance of the final structure of the models that depicts the actual UI. In fact, without detecting the relationship between the GUI elements and their behaviors ' names, it is difficult to construct the PIMs, which as explained later, describe the interaction behaviors between the windows.

Thus, the result of this experiment showed that the PDG analyzer tool does not provide all the details and information that is needed from the PDGs to generate the models. The widgets' names within the Java code and their labels in the GUI were some details not obtained. Moreover, although it can determine the behavior of each widget, it is hard to determine the behavior name of the widgets due to missing information. For example, identifying the *setVisible (boolean)* method can help to define the behavior type of the widget that calls this method (which as mentioned previously, infers an *I_behavior)* but the name of this behavior cannot be defined because there is no information about the *boolean* that is used by this method (whether *true* to open the frame or *false* to hide or close it). The further issue of using this tool is that the hierarchy of all widgets is hard to determine from the PDGs, where no information is available about widget embedding. Due to this issue, the process of determining the widgets that belong to the appropriate PModel is impossible.

To attempt to solve all of these problems, it was decided to combine dynamic analysis methods with this experiment and investigate whether this assists in creating the complete PModels. This is explained in detail in the next section.

5.3 Experiment 3: Combining dynamic and static analysis

As described in the previous section, analyzing PDGs for an interactive system is an effective way to detect the widgets of GUIs, but this method cannot help to extract all the required information for PModels in full. If some important information has been missed from the PDG, for example, it is hard to identify the name of the widgets in the models. Moreover, the PDG analyzer tool is able to extract the type of widget behaviors, whether I_ behaviors or S_behaviors, but it cannot determine the name of these behaviors. Finally, this tool cannot detect the widget hierarchies to create the final structure of the models. To find the information missing from the previous experiment, it was decided to combine the information extracted from both dynamic and static methods to create the full models. The dynamic analysis cannot be depended on because there is a problem with loss of hidden information. Most of the dynamic tools focus only on describing the windows of the GUI system, their elements and structure and interaction behaviors between these windows, ignoring the underlying system functionalities.

Thus, this experiment uses one of the dynamic reverse engineering tools, GUI Ripper[8], to extract the structure for the PModels and then combines both the generated models from GUI Ripper tool and the PDG analyzer tool (described in section 5.2.).

The next section explains the experiment and investigates whether the final information from both dynamic and static methods can help each other to create the required models of the interactive system, and also explores all the problems in this experiment.

[8] http://www.cs.umd.edu/~atif/GUITAR-Web/gui_ripper.htm

Extracting models from GUI Ripper

The GUI Ripper v1.1 Tool is used to run the software's GUI automatically by simulating user actions. It opens all the windows of the software under test to identify and extract all the GUIs widgets, along with their properties, and values. This tool generates a number of models extracted into xml files. The file that contains the GUI's structure information will be examined. This file describes all the information about the windows and their components, and all possible interactions among the events in these components. A textual excerpt of this structure file for the "BMI calculator" application is shown in Figure 5.12.

```
<GUIStructure>
 - <GUI>
   - <Window>
     - <Attributes>
       - <Property>
           <Name>ID</Name>
           <Value>BMI Result</Value>
         </Property>
       - <Property>
           <Name>Modal</Name>
           <Value>false</Value>
         </Property>
       - <Property>
           <Name>Rootwindow</Name>
           <Value>false</Value>
         </Property>
       </Attributes>
     </Window>
   - <Container>
     - <Contents>
       - <Container>
         - <Attributes>
           - <Property>
               <Name>ID</Name>
               <Value>w67</Value>
             </Property>
           - <Property>
               <Name>Class</Name>
               <Value>javax.swing.JFrame</Value>
             </Property>
           - <Property>
               <Name>Type</Name>
               <Value>SYSTEM INTERACTION</Value>
```

Figure 5.12. Part of GUI Ripper tool output for "BMI calculator" program.

GUI Ripper works well with the "BMI calculator" application, but only after adding some modifications, however, it does not work with the other example applications listed in the beginning of this chapter. The problem with this tool is that it does not accept any human intervention during the run. For example, in this application, "BMI calculator", the user must fill in the height and weight, but this tool does not accept such input during the test. So, for testing purposes constant values for weight and height were added in the code to make the program work during the test without any human intervention. Moreover, this tool does not work if there is any action, which quits the program during the test. Thus, to make the example application, "BMI calculator" work well with the GUI Ripper tool, the widgets that terminate the program, such as: ***Close*** and ***Quit*** buttons were deleted.

In this experiment, the structure file was read line by line and all the windows and their widgets and their associated values and the interactive events invoked by these elements were collected. Table 5.7 shows the extracted information that was obtained from "BMI calculator" application.

Window name	Widget type	Widget Label	Widget value	Invoke
BMI Result	JLabel	-	Your Body Mass Index: 23	-
BMI Calculator	JLabel	-	Current weight is:	-
	JTextField	-	-	-
	JLabel	-	Current Height is:	-
	JTextField	-	-	-
	JComboBox	-	. Kilograms . Pounds	-
	JComboBox	-	. Centimetres . Inches	-
	JButton	Calculate	-	BMI Result
	JButton	Clear	-	-

Table 5.7. Information extracted for our example program from the GUI ripper tool output file.

The table above shows that this tool identifies each window and describes the widgets inside it. This helps to detect the widget hierarchies and create the final structure of PModels. However, there was no available information about the labels of some widgets, such as *JTextField* and *JComboBox* . In this case, their types were used as the names of the widgets in the models. Thus, based on this method, the PModel will be as follows:

```
Mbi_Calculater is BMI_CalculatorWin : BMI_ResultWin

        BMI_CalculatorWin is
           (TextField, Entry, ())
           (ComboBox, Container, ())
              (KilogramsItem, SvalSelector,())
              (PoundsItem, SvalSelector,())
           (TextField, Entry, ())
           (ComboBox, Container,())
                    (CentimeteressItem, SvalSelector,())
                    (InchesItem, SvalSelector,())
           (CaculateButton, ActionControl, (I_Result))
           (ClearButton, ActionControl, ())

     BMI_ResultWin is
           (Result, SvalueResponder, ()
```

The model above effectively describes the general structure of the GUI, which depicts the windows and their elements. Moreover, it describes the interaction behaviors that occur in the program. However, it is difficult to identify the system behaviors that occur in the application. Thus, as mentioned before, this experiment is based on combining the dynamic and static analysis methods to create correct and full PModels. So, a comparison of the PModels from the extracted models in experiment 2 (section 5.2) and this one was needed.

Results and problems identified

The extracted models from both PDGs and the GUI Ripper tool outputs are presented in Table 5.8.

A (PDGs)	B (GUI Ripper)
Wind_1 is Wind_2 is (HeightTF, Entry,()) (HeightT, svalueResponder,(S_behavior)) (weightTF, Entry,()) (weightTF, svalueResponder,(S_behavior)) (ComboBox, Selctor,()) (ComboBox, Selctor,()) (Widget,actionControl,(I_Behaivour,S_Behaivour) (Widget, actionControl,(I_Behaivour)) (Widget, actionControl,(S_Behaivour)) (Widget, actionControl,QuitApp)	BMI_CalculatorWin is (TextField, Entry, ()) (ComboBox, Container, ()) (KilogramsItem, SvalSelector,()) (PoundsItem, SvalSelector,()) (TextField, Entry, ()) (ComboBox, Container,()) (CentimeteressItem, SvalSelector,()) (InchesItem, SvalSelector,()) (Caculate_Button, ActionControl, (I_Result)) (Clear_Button, ActionControl, ()) BMI_ResultWinis (Result, SvalueResponder, ()

Table 5.8. The extracted PModels for "BMI calculator" program from **(A):** static analysis (PDGs) and **(B):** dynamic analysis (by using GUI Ripper tool)

Table 5.8 shows that the models built from the static analysis (PDGs) describe both types of behaviors of the widgets effectively but without identifying their behavior names, whereas dynamic analysis is able to detect only the I_behaviors with their names, such as ***I_Result***. So, combining these two sets of information may help, to some extent, in identifying the behaviors for these widgets. Moreover, dynamic analysis helps to detect the widget hierarchies and describe most of the widget names. Table 5.9 summarizes all the extracted information of both types of analysis (PDGs from static code and dynamic analysis using GUI Ripper) in terms of whether it could detect widget names, categories, their behaviors and the widget hierarchies. Thus, based on all of this extracted information, the final models from both (A) and (B) can be more correct and adequate.

Widget	(A) PDGS	(B)GUI Ripper
Name	No	Yes
Category	Yes	No
Behavior :		
Type	Yes	I-behavior
Name	no	no
Hierarchy	no	yes

Table 5.9. Shows the information extracted from each approach

However, it is still difficult to complete the models due to missing information. Where there is no information, it may help if it is possible to link the models extracted from both experiments. For example, there are two *TextField* widgets in (A), which are described as *(HeightTF, Entry,())* and *(weightTF, Entry,()),* whereas the same widgets are described in (B) as *(TextField, Entry, ())* and *(TextField, Entry, ())*. In this example, it is hard to link these widgets and know which widget matches each other. A further example is the *Calculate* button, which is described as *(Calculate_Button, ActionControl, (I_Result))* in (A), but it is hard to identify which one of the buttons described in (B) matches it, whereas in (B) there are two buttons, which have I_behaviors, and are described as (*Widget actionControl, (I_Behavior, S_Behavior)* and *(Widget, actionControl,(I_Behavior).*

Thus, this experiment described in detail our method of extracting the models by using the dynamic analysis tool, GUI ripper, and then trying to apply this information to the extracted models from the static analysis (PDGs). However, we found some difficulties that prevented the creation of full models; where we could not create the correct and full PModels due to missing information from the extracted information from static and dynamic analysis. Where, there is not any information it can help to link the extracted models from both experiments to create the required models, however there are still problems with this approach, as described.

5.4 Experiment 4: Extracting models from direct static source code

Typical Java source code consists of a number of function definitions, variable declarations and statements. This study is only interested in determining the information that is more or less directly related to the GUI library and will focus only on the information needed for generating the models. This part describes in detail the methods that were used in this experiment to extract the widgets and their behaviors from the static code in order to build PModels and PIMs, and discusses some difficulties and limitations that were faced. In the previous sections, the greatest issue found in all the experiments is the difficulty of extracting the full PModels of an interactive application. This section describes the experiments for the extraction of the required information to create the models from the code manually. How to detect the GUIs elements and their related information from source code, then how to detect widget hierarchies is explained in detail; it is very important to group the widgets associated with each window that represents a PModel. Extracting behaviors of widgets from their related events and event handlers are described in this section. This section also introduces the implementation tool to generate PModels automatically. Then the positive and negative sides of this experiment are identified and discussed.

Detecting the GUI elements and relative information

Graphical user interfaces contain a number of widgets such as windows and buttons, which are responsible for capturing interaction from the user. As described above, each window or frame in a GUI program is represented by one PModel, and all the elements inside this window represent the widgets in the PModel. Each widget is described in three parts (name, category, set of behaviors). Therefore, first, it is necessary to detect all the widgets and from these widgets all their related information

can be detected and used to create the requirement models. Thus, the first step is identifying the widgets from the code. To detect the widgets, how they are created in the program needs to be known.

Creating a particular widget requires that a variable of this element is first declared. In Java, all the variables that are expected to be used in a program must have been declared with their specific data type before dealing with them in the code. For data types, to be identified as widgets, they should be in some way related to the GUI library. Thus, determining data type plays an important role in our analysis. This is because if all the data types for GUIs can be found, all variables and parameters that belong to the GUI could easily be identified, and then all the functions and methods that are associated with these variables can also be found.

In a Java program, instantiating widgets must be done by using their constructor. In some cases, factory patterns are used to create an application's widgets. In both ways of creating widgets, widgets' variables and their data types must be identified.

It is certain that detecting widget types helps to identify the widget category in PModels. For example, if the widget type is 'JButton', this means the category is an '***action control'***, or if its type is 'JLabel', the category directly will be a '***display'***. But in the models must be described as a triple (name, category, (behaviors).

For PModels, the names of these widgets as represented in the GUI program must also be identified. As mentioned earlier, a widget constructor must be called when the widget is instantiated. These constructors may have one or many parameters. This parameter could be empty, and in others it represents a value. In some cases, this value refers to another variable or method in the program; we are also interested in this variable type, or tracking this method. In GUIs, the parameter is represented as a

label or title property of the widget, such as with buttons or checkboxes; in other widgets it represents a value such as a string for text fields; or in others it is used for displaying information to the user, such as labels. Thus, understanding the meaning of the parameter will be dependant on the type of each widget; this is because each widget has its own properties and its own uses.

Based on the explanation above, finding the names of widgets for the models is different depending on the widget properties. In fact, it is hard to determine the appropriate names of all the widgets. Consequently, each widget will be dealt with depending on its properties. This means that for the widgets that have labels in the GUI, the widgets' labels are used to describe the name of the widget in the models, such as buttons. On the other hand, the names of the variables in the code are used in the models for widgets that do not have labels, but have values such as text fields, or for widgets that use the value for displaying information to the users, such as Label components. The syntax to write the widget name in a PModel is to write a variable name or widget label depending on widget properties followed by the widget's data type (like *Button* or *Butt*...). For example, the name of a button called ‘close’ can be described as *Close_butt*, *close_button*, *or closebutt*, etc. The intention of the naming convention is to make it easier to relate the PModel to the actual UI.

Examples

Examining some simple code segment statements will illustrate how the widgets and the desired information in each one can be detected, and how the model based on extracted information can be created.

Example 1

```
JButton CloseButton;
JCheckBox GoodCheck;
JLabel text;
CloseButton = new JButton("Close");
GoodCheck = new ();
GoodCheck.setText("Good ");
text = new JLabel (“Enter your name:”);
```

There are three variables in this example: ***CloseButton,*** which is declared as type ‘JButton’, ***GoodCheck*** which is declared as type ‘JCheckBox’, and the ‘***text’*** variable, which is of type ‘Jlabel’. By using these widgets’ constructors, they are instantiated. In this example, the ***CloseButton*** element is created with a one-parameter constructor. This parameter is written between quotation marks, which specify its label as ***Close***. However, the ***GoodCheck*** widget uses the ‘*SetText’* method to specify the checkbox’s label. The ‘*setText’* method is responsible for setting the label for the component. In this example, the ***GoodCheck*** element has been named ‘***Good’***. Finally, ‘***text’*** is created with a parameter. It is known that the parameter in the ‘*JLabel’* widget is used to display values as information for the user in GUIs. Here, the ‘***text’*** component displays ‘***Enter your name***’ in the GUI. The information that can be extracted from this example is illustrated in Table 5.10

Data type	Variable	Label	Value
JButton	CloseButton	Close	-
JCheckBox	GoodButton	Good	-
JLabel	text	-	Enter your name:

Table 5.10. Information, which can be extracted from Example 1 segment statements

Note: the information in the "Label" and "Value" columns describe the parameters as represented in each widget constructor depending on widgets' data type.

From the information above, the first part of the PModels can easily be created. Where *Close* button represents the label of the ***CloseButton*** variable; in this case ***Close_Button*** is written as the widget name in the model. Similarly, ***GoodCheck*** Checkbox is labeled as ***Good***, and then in the model will be labeled as ***Good_CheckBox*** or ***Good _Check***. However, the widget 'text' does not have a label due to its properties; its parameter represents as a value, in this case, as described earlier, the name of the variable in the code is used as a name in the PModel, to be ***text _Label***. Moreover, the widget category can be identified from the widget type, where for every Swing widget there is a known related category. Thus, from the example above, the categories of '*JButton'* and '*JCheckBox'* are ***ActionControl***, whereas the category of '*JLabel'* will be ***display***. Therefore, the PModels of the example above will be:

(Close_Button, ActionControl, (…))
(Good _CheckBox, ActionControl, (…)) } Using widget's label as a name

(text_Label, display, (…)) → Using widget's variable name

Example 2

There is a data type that does not belong to the GUI library, but it is used with one of the elements that does belong to this library. This data is of interest. The following example is used to clarify this:

```
String W[] = {"kilogram","pound"};
JComboBox WCB = new JComboBox(W);
```

This example declares two variables: (1) "W" and its type "String Array"; and (2) "WCB" and its type "JComboBox". As described above, all the variables that should be part of the GUI are of interest. This example has one combo box, which is created with a one-parameter constructor to specify the value of the drop-down list of this widget. Here, the items of the drop-down list are taken from the array "W", which contains the strings "kilogram" and "pound". In the PModels, the comboBox is treated as a container of the values of the drop_down list. These values represent the contents of the array "W" in this example. Therefore, it is important to know the information of any other type that does not belong to the GUI library, such as String, int etc. in cases where this data has a relation with any element belonging to the GUI library. All the information that is obtained from this example is shown in Table 5.11 below.

Data type	Variable	Label	Value	Related to
JComboBox	WCB	-		-
String	W	-	Kilogram, pound	WCB

Table 5.11. Information, which can be extracted from Example 2 segment statements

Sometimes, each element in the drop-down list has its own event. This means that each element may have its own set of behaviors. Depending on this information, the PModels will be:

(WCB_Sel, Container, (...))
    (Kilograms_Item, SvalSelector,(...))
    (Pounds_Item, SvalSelector,(...))

Example 3

In Java, classes may inherit from another class, where a class can be declared as a *subclass* of another class using the "extends" keyword, followed by the name of the class to inherit from. Variables can then be declared as a type of this subclass, which in some cases may be subclasses of widget classes. These variables are also of interest. The following code segment illustrates this.

```
public class MyButton extends JButton
 {
     .
     .
    public MyButton(String label)
   {

      super(label);

   }
     .
     .
     public static void main(String[] args) {
        MyButton button1 = new MyButton("OK");
        MyButton button2 = new MyButton("Cancel");

        .
        .
 }
}
```

Here, is a "***MyButton***" class that inherits from '*JButton*' is created, and then the variable "***button1***" and "***button2***" are declared as type of "***MyButton***" class. The constructor "***MyButton***" in class "***MyButton***" needs one String type parameter to be used as a label of the widget (see Table 5.12).

Data type	Variable	Label
MyButton	Button1	OK
MyButton	Button2	Cancel

Table 5.12. Result of extracting widgets from Example 3.

Describing the widgets in a PModel for this example will give:

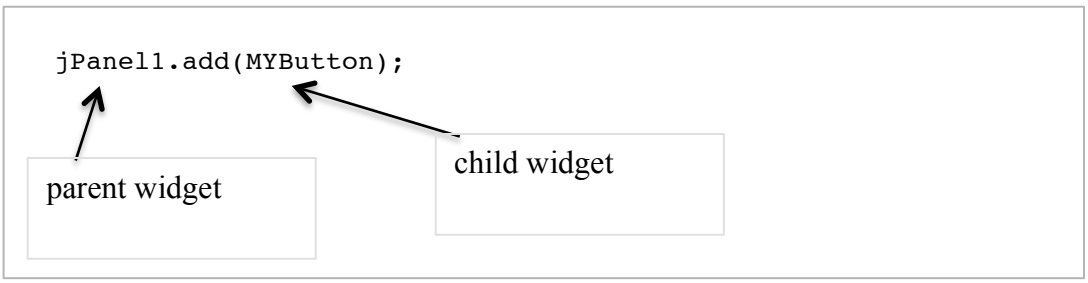
(OK_Button, ActionControl, (...))
(Cancel _CheckBox, ActionControl, (...))

This step of the analysis shows how to extract the widgets from code and determine two parts of PModels of each widget: (1) name, by identifying the widget's label or widget's variables depending on each widgets properties; and (2) category by identifying the widget type that belongs to GUI library. The next step will illustrate how widget hierarchies in GUI applications can be determined to create the PModel's structure.

Detecting widget hierarchies

The relationship between one widget and another composes a tree called the 'widget hierarchy'. For instance, if there are a number of buttons in a 'panel' that belongs to a 'window', these buttons are represented as child nodes of the panel node, which itself is a child node of the window node. Once all of the widgets have been identified in the previous section, the connections between these widgets needs to be detected. Defining the widget hierarchies is very important in order to collect all of the widgets that belong to each window or frame in the system to set up the final structure of the models, where, determining the widget hierarchies helps to identify the affiliation of each widget to its own model. As explained previously, each window or frame is represented as a PModel and each widget belongs to this window, and each widget belonging to these windows will be described in the model as a list of triples (a name of the widget, category, (set of behaviors)).

To detect widget hierarchies, the code is searched for the expressions that embed one widget into another. In Java, widget embedding can be done directly with the widget construction, by using a particular method, e.g. the '*add* ()' method is used to add a child widget to the parent widget. Consider the following statement:



To identify the embedding widgets that are developed by widget construction, any statement that contains the '*add* ()' method in the static code is inspected, and then the

parent and the child widgets are identified. A part of the “BMI Calculator” example is shown as clarification of this.

```
public class MBI_calculater extends JFrame {
    public MBI_calculater()
    {
      JFrame frame2 = new JFrame("Result");
      JPanel jPanel1 = new JPanel();
      jPanel2 = new JPanel();
      JLabel Titel= new JLabel();
      weightLabel = new JLabel("Current weight is :");
      jPanel1.add(weightLabel);
      weightTF = new JTextField();
      jPanel1.add(weightTF);
      weightLabel1 = new JLabel();
      weightLabel1.setText("e.g: 58 kg  or 128 pound");
      jPanel1.add(weightLabel1);
      JComboBox WCB = new JComboBox();
      HeightLabel = new JLabel("Current Height is :");
      jPanel1.add(HeightLabel);
      HeightTF = new JTextField();
      jPanel1.add(HeightTF);
      HeightLabel1 = new JLabel();
      HeightLabel1.setText("e.g: 64 inches  or 160 centimeters");
      jPanel1.add(HeightLabel1);
      final JComboBox HCB = new JComboBox();
      jPanel1.add(HCB);
      JButton ClcBtn = new JButton("Calculate");
      jPanel1.add(ClcBtn);
      JButton closeButton = new JButton("Close");
      jPanel2.add(closeButton);
      JButton ClButton = new JButton("Clear");
      jPanel1.add(ClButton);
      JButton quitButton = new JButton("Quit");
      jPanel1.add(quitButton);
      add(jPanel1);
      frame2.add(jPanel2);

  public static void main(String[] args)
     {   MBI_calculater BMI_mainWin = new MBI_calculater();
         BMI_mainWin.setTitle("BMI Calculator");
        .
        .
        .
     }
 }
```

In this example, a number of widgets are created and then the “*add* ()” method is used to embed a widget into another one. Notice that no event is added to the widgets here, this is to keep the example very simple in order to understand this step of analysis. In the beginning, the process described in the previous section was followed to identify all of the widgets in the example and then each statement was checked to see if it

contains the widget embedding method to identify the parent widget of each child widget. In this example, the information that can be extracted is illustrated in Table 5.13, where "Location" column represents the parent widget of each child widget identified from the "add () " method.

Type	Variable	Label	Value	Location
JLabel	weightLabel	-	Current weight is:	JPanel1
JLabel	weightLabel1	-	e.g.: 58 kg or 128 pound	JPanel1
JLabel	HeightLabel	-	Current weight is:	JPanel1
JLabel	HeightLabel1	-	e.g: 64 inches or 160 centimeters	JPanel1
JTextField	weightTF	-	-	JPanel2
JTextField	HeightTF	-	-	JPanel1
JComboBox	WCB	-	-	JPanel1
String[]	W		Kilogram, pound	WCB
JComboBox	HCB	-	-	JPanel1
String[]	H	-	inches, centimeters	HCB
JButton	ClcBtn	Calculate	-	JPanel1
JButton	ClButton	Clear	-	JPanel1
JButton	quitButton	Quit	-	JPanel1
JButton	closeButton	Close	-	JPanel2
JLabel	Titel	-	>> your Body Mass Index:	JPanel2
JPanel	JPanel2	-	-	frame2
JPanel	JPanel1	-	-	BMI_mainWin
JFrame	frame2	Result	-	-
MBI_calculater	BMI_mainWin	BMI Calculator	-	-

Table 5.13. Eextracted widgets and the parent widget of each child widget from embedding method from Example 3

Using this information, the widget hierarchy for this example is shown in Figure 5.13.

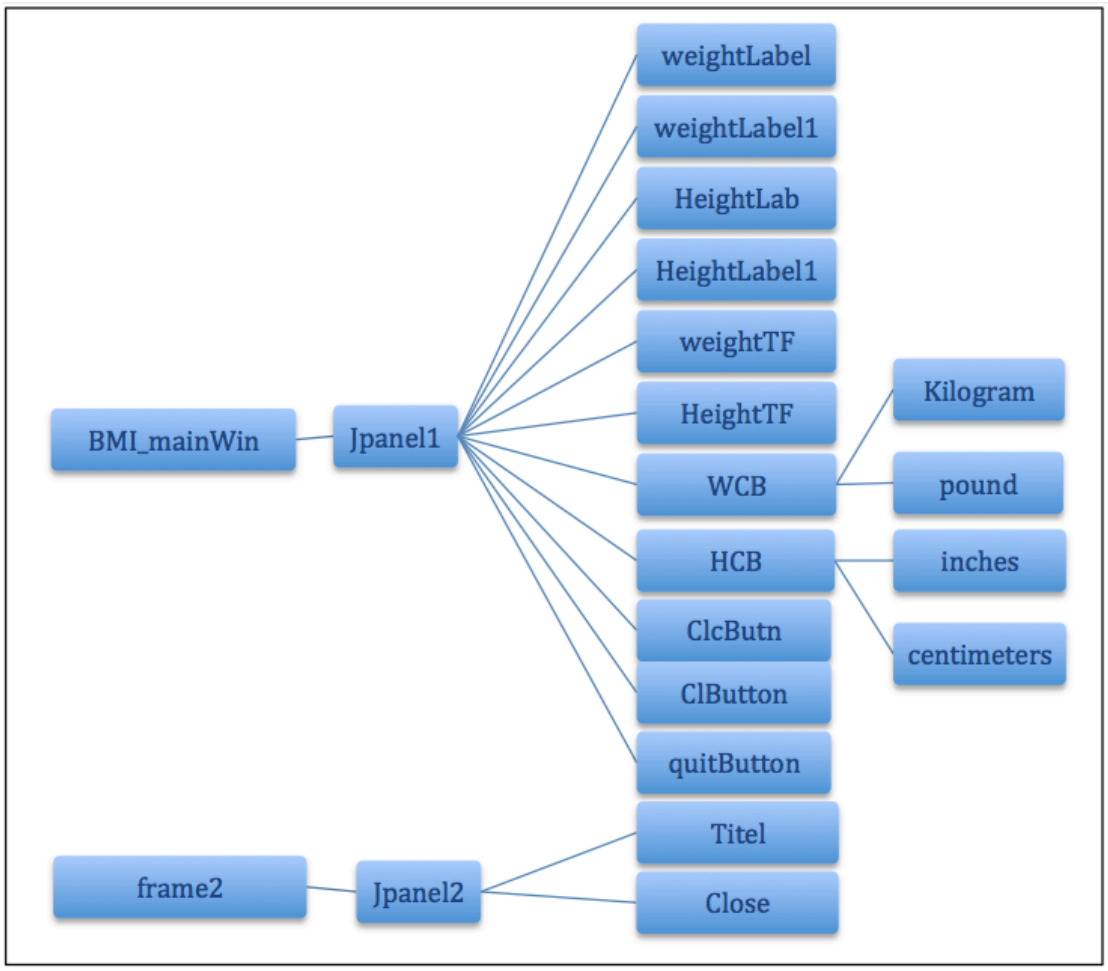


Figure 5.13. Widget hierarchy based on the variable names of the widgets in Table 5.13. (Note: Each element in the array is represented in a separate node).

Consequently, all the widgets in any application can be combined to easily create the final structure of PModels. In the example above, there are two PModels, (1) BMI_mainWin, whose title is "BMI Calculator" and (2) frame, whose title is "Result". Each Model has a number of widgets that belong to it and each of these widgets is described by the name, category, and behaviors triple. Thus, based on information extracted, the PModels for this example are:

```
BMI Calculator_Window:
        Jpanel1, container, (...))
          (weight_Label, display, (...))
          (weight_Label1, display, (...))
          (WeightEntry, Entry, (...))
          (WCB_Sel, Container, (...))
              (Kilograms_Item, SvalSelector,(...))
```

(Pounds_Item, SvalSelector,(...))
(Height_Label, display, (...))
(Height_Label1, display, (...))
(HeightEntry, Entry, (...))
(HCB_Sel, Container, (...))
(Centimeteress_Item, SvalSelector,(...))
(Inches_Item, SvalSelector,())
(Caculate_Button, ActionControl, (...))
(Clear_Button, ActionControl, (...))
(Quit_Button, ActionControl, ((...))
Result_Window is
Jpanel2, container, (...))
(Titel_Label, display, (...))
(Close_Button, ActionControl, (...))

According to the information above, the inferred structures of two windows are as shown in Figure 5.14. Comparing this Figure with real screenshots for the "BMI Calculator" windows (Figure 2.2 in section 2.2), it can be seen they both have the same number of frames and widgets. However, nothing can be determined about the position of the widgets or the layout within the windows from the model (which is intentional as this is abstracted out of the models).

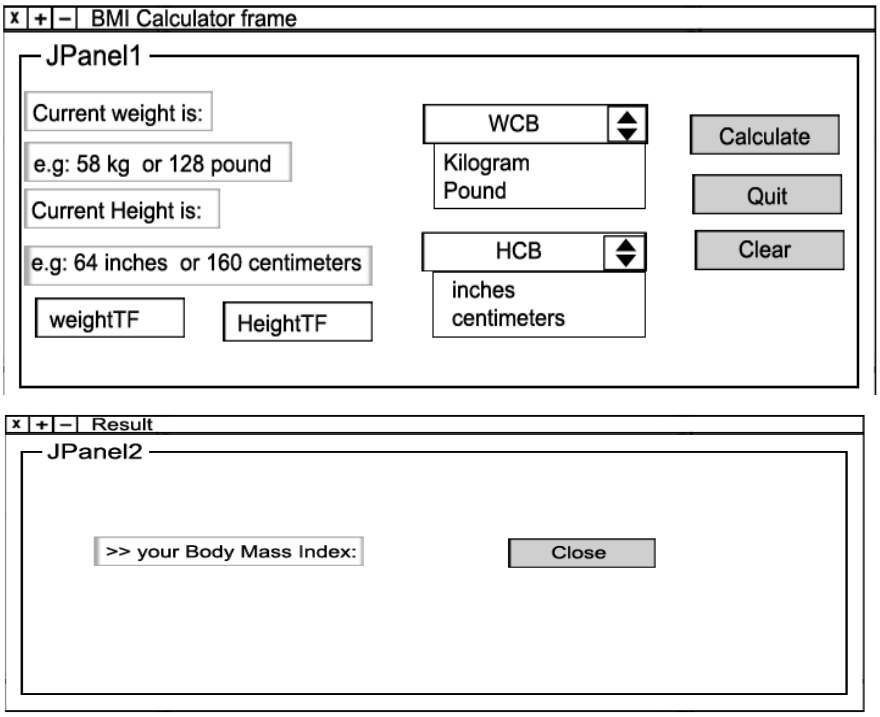


Figure 5.14. Overall structure of the windows of the example based on the information in Table 5.13

Detecting widget hierarchies is an important analysis step to create the overall structure of the models. With this step, all of the child widgets can be tied to their parent widgets to determine the structure of the models as shown in the graphical user interface of the application. Once the relationship between the widgets has been identified, it becomes easy to detect each window and all the widgets that belong to it to generate the models. The next step will show how the behaviors of each widget can be determined to create the complete PModels and PIMs of the interactive programs.

Detecting widget behaviors

The behavior of the widget describes the action or event that occurs when the user interacts with this widget. This event connects with event handlers of each widget through event listeners. A listener listens for specific user interface activities and then implements the code that is related to those activities. Thus, detecting the widget events will be the starting point to identifying the widget behaviors. This section will examine detecting the connections between events and their event handler, where the event handler can be a function. Then, the widget behavior from the code inside this event handler can be detected.

Detecting event connections

Every single component or widget can generate more than one type of event. A button for example, can generate a Mouse Event and an Action Event. To get an event processed, there are two things, which must be in the code to implement events [Gehringer01]:

- "Register an event listener, and
- Implement an event handler".

In Java, a suitable listener is usually registered on each widget that it is interested in, and then the relevant event handler of the widget is implemented. Table 5.14 shows some examples of user interface events and their required type of listeners and relevant methods.

Event	Listener	Register to element by	Relevant method
ActionEvent	ActionListener	addActionListener method	- actionPerformed
FocusEvent	FocusListener	addFocusListener method	- focusGained
			- focusLost
ItemEvent	ItemListener	addItemListener method	itemStateChanged

Table 5.14. Some events and their related listeners.

The simple code segment that defines a button and processes its event illustrates this.

```
JButton MYButton = new JButton("reset");

MYButton.addActionListener(new ActionListener()

    public void actionPerformed(ActionEvent e)
    {
        ...
    }
```

When the button is pressed and released:

The *ActionListener* receives events from *MYButton* button. The component's *addActionListener* method is used in order to register the Listener to the item *MYButton*. The *ActionListener* interface has one method *actionPerformed*, which is the event handler. Thus, when the action event occurs, the event will be picked up and passed to the *actionPerformed* method and then the code inside will be executed. Thus, the important data of interest in this example is:

- Registering the event Listener *ActionListener* to the widget occurs by using the *addActionListener* method;
- The event for this widget is *ActionEvent*; and
- The event handler is the *actionPerformed* method, from which the behaviors need to be detected.

Examples

Three different methods to extract widgets' events will be examined. To keep things simple, in these examples a button is created that quits the system when it is clicked. The *ActionListener* interface has one method *actionPerformed* that makes the system call *System.exit(0)* to quit the application.

Method 1

```
public class QuitButton extends JFrame
{
    public QuitButton ()
    {
        JButton quitButton = new JButton("Quit");

        quitButton.addActionListener(new ActionListener()
            {

                public void actionPerformed(ActionEvent e)
                {

                   System.exit(0);

                }
            });

        getContentPane().add(button);

      }

    public static void main(String[] args){

       QuitButton frame = new QuitButton ();

        frame.pack();

        frame.setVisible(true);

      }

}
```

Here, the button *quitButton* in the *QuitButton* constructor is declared and initialized. Next, the *ActionListener* interface is provided inside the *ActionListener* method to register the action listener with the button. Finally, whenever the button is selected, the *actionPerformed* method is called to handle the event. From the code inside the *actionPerformed* method, the behaviors of the widget can be determined. This is explained in more detail later in this section.

Method 2

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class QuitButton extends JFrame implements ActionListener
{
    public QuitButton ()
    {

      JButton quitButton = new JButton("Quit");

      quitButton.addActionListener(this);

      getContentPane().add(quitButton);

    }

    public static void main(String[] args)
    {
        QuitButton frame = new QuitButton ();

        frame.pack();

        frame.setVisible(true);
    }

    public void actionPerformed(ActionEvent e)
    {

        System.exit(0);

    }
}
```

First, the *ActionListener* interface is implemented in the ***QuitButton*** class. The *actionPerformed* method is used in this class to implement the listener interface. Then, the *addActionListener* method is used to register the listener with our button. Similarly to the first method, when the button is clicked, the *actionPerformed* method is called to handle the event.

Method 3

```
public class QuitButton extends JFrame
{
public QuitButton()
    {
        JButton quitButton = new JButton("Quit");

        quitButtonListener buttonListener = new quitButtonListener();

        quitButton.addActionListener(buttonListener);

        getContentPane().add(quitButton);
    }
    public class quitButtonListener implements ActionListener
    {
      public void actionPerformed(ActionEvent e)
        {

            System.exit(0);

        }
     }
    public static void main(String[] args)
    {
        QuitButton frame = new QuitButton();

        frame.pack();

        frame.setVisible(true);
    }
}
```

A third approach is to create an inner class called ***quitButtonListener***, which implements the *ActionListener* interface. This class contains only an *actionPerformed* method. The declaration and initializing of our button will be in the "***QuitButton***" constructor. After that an instance of the ***quitButtonListener*** class is declared and then this listener object is registered to this button by using *addActionListener* method. Consequently, when the button is clicked, the *actionPerformed* method is called to handle the event.

The three examples above show that creating events and adding them to the widget is comprised of two parts: (1) register an event listener for the element; and (2) implement the listener interface. From this it can be inferred that whichever way the

code is written, these two things are crucial for getting the event processes of any widget.

The algorithm that is used to find out the event and event handler of an element in order to identify the behaviors of this element is described in the following section. In the beginning, the method that is responsible for registering an event listener method is searched for (see Table 5.14, for examples). Next, the event listener interface is tracked, to discover whether it is provided from inside the register method or from a class that implements this listener. Next, the event handler that this listener connects with is looked for. Finally, from the code inside the event handler the behaviors of the widget can be identified.

Detecting behaviors

Having found the events and the event handlers, we can determine the behaviors of the widgets. To detect behaviors, the code in the event handlers must be read line by line and each statement inspected to identify the type of behavior to identify whether it is an S_behavior, I_behavior or both. Two important things are explained in this part: (1) how to detect the behavior type of each widget; and (2) how to identify the behavior names that describe the occurring behaviors of the widgets.

I_behaviors navigate the windows of the system by using special methods that affect the window's interactions, such as *setVisible* method. The names of interaction behaviors identify the interaction that happens. For example, any component calls the *setVisible* method for one of two purposes, either open or close the window. This occurs depending on the value of Boolean parameter of this method, where if this parameter was "*true*", this method shows or opens this window and then the name of this behavior will be *"I_Open"* or *"I_Show*". In contrast, if it was "*false*", the

window will be closed or hidden and then the name of this interactive behavior is "I_*Close*" or "*I_Hide*". Hence, being able to identify the functions that define I_behaviors, it is also easy to determine the expected name of this behavior (see Table 5.6 (in section 5.2) for more examples about methods that help to detect the interaction behaviors).

An S_behavior is related to some operation of the system specification or state operation. Although there are some static methods that can help to infer S_ behaviors, such as *setText* and *getText*, it is difficult to limit all the expected statements or methods that may help to identify S_ behavior from the code. Not only it is hard to identify the functions that have S_ behaviors, it is also hard to determine the name of this behavior. How to name the S_ behaviors will be further explained in detail with some examples, later in this section.

To detect behaviors, each statement inside the event handler needs to be checked. If any statement belongs to any one of the I_ behavior groups this means this behavior is an I_ behavior. However, if any statement belongs to the S_ behavior group this means that this behavior is S_ behavior. Moreover, if there two statements and one of them belongs to the I_behavior group, and other one to the S_ behavior group, then this widget has multiple behaviors: S_ behavior and I_ behavior. Finally, if there is no statement in the event handlers belonging to any one of these behaviors, this widget maybe not have any behavior or maybe is used to quit the application if it has "*system.exit(0*)" statement.

It is also possible that there is an anonymous function call inside the event handler, where this function is created by the programmer. This is named an anonymous function, because they are declared inline in the code with no method name, which

means they cannot be identified as either I_ behavior or S_behaviors. To solve this problem, each statement inside this procedure also needs to be inspected to identify whether there is any statement that belongs to interaction or system behavior. The algorithm to identify the behaviors from the code inside the event handlers can be summarized in Figure 5.15

```
Check each phrase (P) in the event handler:

if  P ∈  I_ behaviors  set
        Put I into behaviors  _type
if P ∈ S_ Behaviors  set.
        Put S into behaviors  _type
if  P ∈ Procedure Calls
     Go to check each statement inside this procedure whether ∈ I_ behaviors set, or ∈ S_
     Behaviors set or ∈ both sets, and put the result into behaviors  _type
if P ∈ Quit
        put Q into behaviors  _type
begin
       forall I ∈ behaviors  _type
         count_I++
       forall S ∈ behaviors  _type
         count_S++
       forall Q ∈ behaviors  _type
         count_Q++
       if(count_Q>0)
            Behavior  ="QuitApp";
         else
         if((count_S>0)∪( count _I>0))
           Behavior  ="(I_ Behavior, S_ Behavior)";
         else
         if(count_S>0)
           Behavior  ="(S_ Behavior)";
         else
         if(count _I>0)
           Behavior  ="(I_ Behavior)";
         else
         if((count _S=0) ∪ (count_I=0))
           Behavior  ="()";
end
```

Figure 5.15. The algorithm used to identify the behaviors from source code

Examples

This is now applied in some segments of our “BMI Calculator” example and how the behaviors of the widgets and other related information from the code inside the event handlers can be detected is explained for each part.

Example 1

```
cJButton closeButton = new JButton("Close");
 closeButton.addActionListener(new ActionListener()
  {
     public void actionPerformed(ActionEvent event)
       {
              frame2.Visible(false);
        }
     });
```

As explained earlier in this section, the event handler of this widget is ***actionPerformed***. By reading each statement in the code inside this method and searching for the behaviors, this line is found:

frame2. Visible(false);

“*Visible*” belongs to I_ behaviors group (Table 5.6 in Section 5.1). This means that the behavior type of "***close***" button is an interaction behavior. The "*Visible*" method is called from the "***frame2***" variable with a parameter "*false*". That means the action of this widget is close "***frame2***" window. Thus, the behavior name of this element will be "***close***”, and so this behavior is written as I_close, and then the complete triple of this widget will be:

(Close_Button, ActionControl, (I_close))

Example 2

```
JButton ClButton = new JButton("Clear");
ClButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
           weightTF.Text(" ");
           HeightTF.Text(" ");
      }
   });
```

In this example, there are two statements that belong to S_ behavior; in this case deals with only one of them. This is because at least one phrase of any type of behaviors (Interaction or System) is enough to determine the type of behavior of the widget. In this event handler, there are "***weightTF***" *JTextField* and "***HeightTF***" *JTextField* which call the procedure "***Text***". This method belongs to S_ behaviors group (tale 5.6 in Section 5.1), and therefore the behavior type of this button is a System behavior.

However, the problem is that the behavior name of this widget cannot be identified. This is because "***Text***" method can be used in multiple tasks, such as setting a label of a widget (as a button) or setting a value of other widget (as *JTextField*). In this case, the widget label will be used rather than the name of behavior. In this example the widget label is "***Clear***" and then the PModel will be:

(Clear_Button, ActionControl, (S_Clear))

Example 3

```
JButton ClcBtn = new JButton("Calculate");
ClcBtn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
    Calculater_fun();
    }
 });
```

Here, the "***Calculater_fun()"*** procedure is called inside the event handler. This is an anonymous function and so it does not belong to I_ behavior or S_ behavior groups. In this case, all the statements inside this function need to be checked to discover whether any of them belongs to one of I_ behavior or S_ behavior groups. Consider the following fragment code, which represents the code of the "***Calculater_fun()***".

```
public void Calculater_fun()
{
    frame2.setVisible(true);
    Titel.setText(" ");

    if(  str2 == H[0])
      {
       temp2 =
          (double)(Double.parseDouble(HeightTF.getText())* 2.54);
       t2 = (temp2 * 0.01);

      } else
      {

          t2 = Double.parseDouble(HeightTF.getText()) * 0.01;
      }

       if(  str == W[1])
       {
          t1 = (Double.parseDouble(weightTF.getText()) / 2.2);
       }else {

               t1 =  Double.parseDouble(weightTF.getText());

             }

   temp1=t2*t2;
   res= t1 / temp1;
   Titel = new JLabel(res );
   Titel.repaint();
}
```

From the segment above, each statement can be read and compared with behavior groups. There are four statements; one belongs to I_ behavior and the others to S_ behavior. Table 5.15, shows these statements with their behavior types and expected behavior name.

Statement	Behavior	
	Type	Name
frame2.setVisible(true)	I_ behavior	Open
Titel.setText(" ")	S_ behavior	Cannot identify
weightTF.getText()	S_ behavior	Cannot identify
HeightTF.getText()	S_ behavior	Cannot identify

Table 5.15. Statements with behavior types and expected behavior names.

As shown in Table 5.15, "***frame2***" calls "*setVisible*" method with a "*true*" parameter. That means when this function is called, "***frame2***" will open directly, and then it can be said that the behavior name is "*open*". Another statement belongs to the S_ behavior group. As explained above, when there is more than one statement

belonging to the same type of behavior, it is enough to say this function also has S_ behavior. However, it is hard to identify the name of this behavior. So, in this case the function name can be used as a behavior name or the widget's label as a behavior name. Which one is the best to be used as a behavior name should be considered.

When the programmers create a GUI application, they often identify the widgets' labels that fit with the expected occurring event when the user deals with this element. For example, if there is a button, which closes a frame when it is clicked on, this button's name will often be "*Close*". However, the function name sometimes is determined as a specific task, which is implemented when it is called, but this is dependent on each programmer's style. This is because some programmers choose convenient names that fit the purpose of this function and some of them understand their code and do not need to explain the purpose of this function to anyone. Hence, we specify the name of the behavior on the basis of the label's widgets, which may be better than the function name. In the example, the name of the S_ behavior, if taken from the widget label will be "*Calculate*". Thus, the final PModel of the widget in this example will be:

(Calculate_Button, ActionControl, (I_Open, S_Calculate))

Thus, detecting the event and event handler of an element plays an important role in this analysis in order to identify the behaviors of this element. As explained, determining the behaviors will be performed by inspecting each expression inside the event handler method and then identifying from the behavior whether it is an interaction or system behavior or both.

Summary

This section analyzes an interactive application's code to extract the widgets and their behaviors to create PModels and PIMs. From a variable data type, all the widgets that are created in any application can be identified, and then it will be easy to detect other related information such as the widget's name and functions, for creating the PModels. As explained, the widget can be detected from the data type, which helps to identify the category of the item in the model, and the name of the widget from the widget's label.

Detecting widget hierarchies is explained in this section. The relationship between widgets was determined, which contributes to creating the overall structure of the models. This will be done by searching for all the embedding widget actions in the code and then it is possible to identify the widget parents and widget children to connect all the windows with their related widgets to construct the PModels structures.

This section also tried to detect all behaviors of each widget from the static code. Where widget behaviors are defined by finding the widget's event and the relevant event handlers, and then the behaviors can be detected from the code inside these handlers. In other words, behaviors will be gathered by checking each statement inside the event handler and identifying whether there is any statement, which belongs to interaction or system behavior.

Implementation

Based on this methodology, a fully automated PModels creation tool was implemented; NetBeans ® 7.3 was used to implement this system. The current implementation of this tool recognizes all GUI elements, widgets hierarchies and behaviors of these widgets. However, this tool does not work for all GUI applications. This section explains in detail how this tool was implemented, and discusses the problems faced in this experiment

The previous section explained all the different phases to extract the widgets and their related information to build the models. These phases focus on the following algorithm to detect the GUI parts:

(1) Detect all the data types of the names declared in the program, such as names of variables, parameters, classes, fields, methods, that should be more or less directly related to the GUI.

(2) Detect the register event listener methods to detect the event handlers.

(3) Detect the widget hierarchy.

To implement all these separate phases in one algorithm, three steps need to be followed: scanner, parser and extracting model.

Scanner step: In this step, each statement in the code will be read and separated into tokens. All white space and comments between tokens are removed from the program.

Parser step: The tokens contain important key words, such as public, class, static, {, }, and so forth. These key words help to create a special algorithm to identify the variables and their types, classes, functions information, and so on. This step organizes the tokens and checks

whether they conform to the syntax of the sequences that are defined by our algorithm, and then, all the important information is extracted and sorted into one table, which contains all the identifiers, and all related information in the program. Each entry in this table contains:

- Name of an identifier
- Type of name (e.g. variable, function, class, field, etc.)
- Type (int, String, Button, etc.)
- value
- ID

ID represents a 'unique number' or is a pointer to another name of an identifier. The unique number of the IDs are used as sequence numbers, where each file in the application has a unique number and in turn, each identifier name in each file has a unique number that comprises the file's ID and its distinctive number separated by '_'. This technique is used to avoid similar identifier names in the application, where there are variables, which may have the same name in different scope in the program. Figure 5.16, shows examples of the method used to number the ID based on the scope.

ID is used as an identifier name in case this identifier calls any one of the astatic methods, such as '*add* ()', '*setText* ()', '*setVisible (boolean*)', etc.). To further clarify, the ID of the statement '(jpanel1.add (CloseButton)', is: ***jpanel1***

Extracting model step: This step checks all the information in the table to see whether it conforms to the syntax of the sequences that are defined by our algorithm (as shown in Figure 5.15) to create the final models.

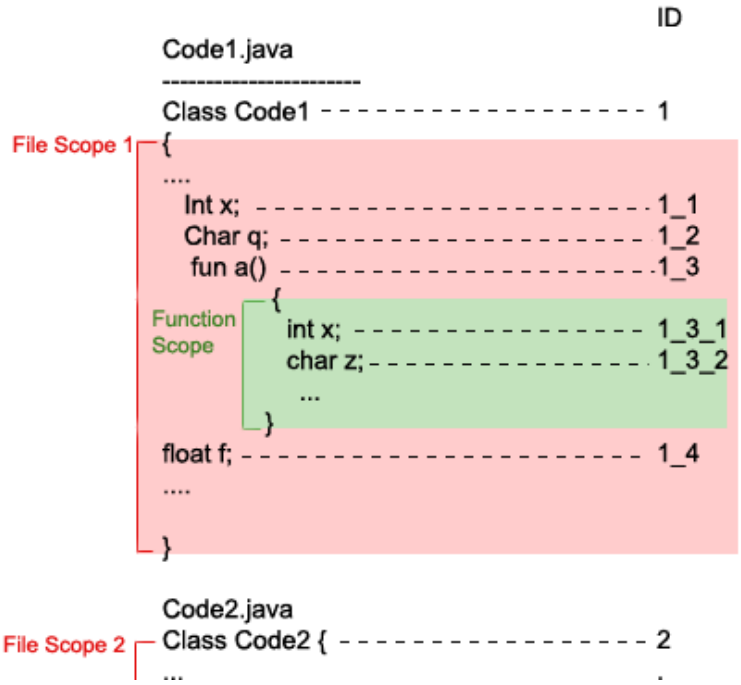


Figure 5.16. Our method to give the ID number of an identifier based on the scope

Results and problems identified

The GUI analysis tool has been implemented as described in the previous sections of this experiment. The purpose of this tool is to extract all the GUIs' widgets and their behaviors, and then from the extracted information, it can generate the PModels automatically. This implementation was run and tested with some of the example code listed earlier in chapter 4, all of which use Java Swing as GUI library. It was found that our tool works with these interactive examples, with both positive and negative aspects.

More precisely, our GUI analysis tool is able to extract almost all information about elements from source code effectively into one table; this information represents an element's name, value, kind (whether it's a variable, class, static method or function), and Java type, (e.g. *String*, *JButt*on, etc.). Figure 5.17, for example, shows a part of the required information that is extracted from "BMI calculator" application into one table.

ID	Name	Kind	type	value
[1]	[BMI_ex]	[Clss]	[_]	[_]
[1_1]	[main]	[fun]	[String[]-> void]	[_]
[1_1_1]	[BMI_mainWin]	[val]	[MBI_calculater]	[_]
[2]	[MBI_calculater]	[Clss]	[JFrame]	[_]
.				
.				
[2_7_1]	[jPanel1]	[Val]	[JPanel]	[ ]
.				
.				
[2_11]	[H[]]	[val]	[String[]]	["inches", "centimeters"]
[2_12]	[HCB]	[val]	[JComboBox]	[H]
[2_15]	[ClcBtn]	[val]	[JButton]	["Calculate"]
.				
.				
[3_22]	[ClButton]	[val]	[JButton]	["Clear"]
[3_23]	[quitButton]	[val]	[JButton]	[Quit"]
.				
.				
[BMI_mainWin]	[setTitle]	[Method]	[_]	[("BMI Calculator")]
[BMI_mainWin]	[setVisible]	[Method]	[_]	[(true)]
[jPanel1]	[add]	[Method]	[_]	[ClButton]
.				
.				

Figure 5.17. An extracted table part of "BMI calculator" app.

This extracted information can help to identify GUI elements, and the widget hierarchies, and then from this table, and the following described algorithm (5.15), all the widget behaviors can be identified. As explained earlier, identifying GUI

elements, the widget hierarchies, and the behaviors of any applications contribute to creating PModels. A textual excerpt of the PModels that are created from the "BMI calculator" application is presented in Figure 5.18.

```
MainWin is
      (weightEntry, Entry, ())
      (WightEntry, SvalueResponder, (S_cleare))
      (WeightEntry, Container, ())
              (KilogramesItem, SvalSlector, ())
             (PoundsItem, SvalSelctor, ())(HightEntry, Entry, ())
      (HightEntry, SvalueResponder, (S_cleare))
      (HightEntry, Container, ())
            (CentimetersItem, SvalSelctor, ())
             (InchesItem, SvalSelctor, ())
       (CalculateButton, ActionControl, (S_result, I_Calulate))
       (ClearButton, ActionControl, (S_cleare))
       (QuitButton, ActionControl, (QuitAPP))

Result_Win is
        (result, SvauleResponder, (S_result))
        (CloseButton, ActionControl, (I_Close))
```

Figure 5.18. Output PModels for "BMI calculator" example.

Once all I_behaviors of the widgets have been derived automatically in the PModels above, PIMs can be generated easily manually. Based on the above output PModels, the PIMs is as shown in Figure 5.19

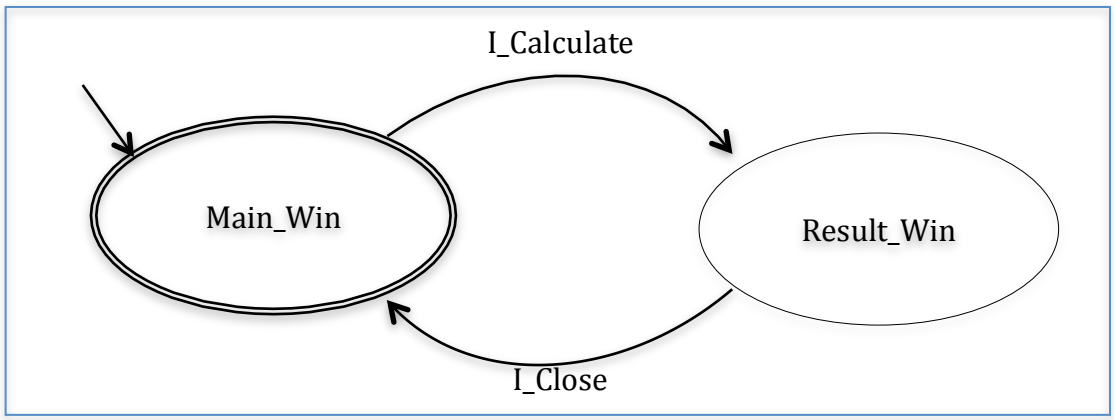


Figure 5.19. PIM for "BMI Calculator" app from the PModels that are shown in Figure 5.18.

However, this tool cannot extract all of the necessary information such as widget name, and the behaviors of some widgets in some examples of interactive applications; these problems in turn stand as obstacles to generating the full PModels. Table 5.16 summarizes and clarifies the aspects that can be extracted from each example, where (✓) represents the features that have been extracted from code for each program by the GUI analysis tool and (✗) represents the features that have not been extracted.

Program name	GUI elements	Element's information	Widget hierarchy	Widget Behaviors
BMI calculator	✓	✓	✓	✓
GoGrinder	✓	✓	✓	✓
Digital Parrot	✓	✗	✓	✗

Table 5.16. Aspects extracted for each example.

As shown in Table 5.16, this tool is able to extract all the GUI elements and widget hierarchies for all interactive examples. However, it is not able to extract the behaviors of the elements in some of the applications. Thus, this tool can generate the required models for some interactive examples, but not for some others. The reason for this problem is that our GUI analysis tool extracts the information based on the set of rules that define the symbols, such as variables, classes and functions, etc., that are considered to be fragments in the Java language. This is effective for extracting all the variable types and identifying the widget hierarchies, however, this tool does not capture all the ways symbols can interact in expressions. This problem makes it hard to understand and extract labels and behaviors of some widgets. A part of the "Digital Parrot" application is an example, which will clarify this.

```
.
JToggleButton trailButton
     = setupNavigatorButton(trailNavigator.getNavigatorName(),
       trailNavigator.getAcceleratorKey(),
       trailNavigator.asJComponent());
    navigatorsBar.add(trailButton);
.
```

In the example above, there is the '***trailButton'*** variable, which is declared as a '*JToggleButton'*. The value of this widget is '***setupNavigatorButton'*** function, which sends the widget information and the event handler found inside this function. But the GUI analysis tool cannot understand all of these processes. As explained earlier, without identifying the event handler the behavior of the widgets cannot be determined. For the same reason, this tool also cannot identify the label of some of the elements. In the following example:

```
private static final String APP_TITLE = "The Digital Parrot";
.
.
JFrame timelineFrame =
new JFrame(timelineNavigator.getNavigatorName()+"_"+ APP_TITLE);
.
.
```

There is a '*JF*rame' variable, which is '***TimelineFrame***' and is declared as type '*JFrame'*. By using these widget constructors, it is instantiated, with one parameter. This parameter has a composite value that consists of '***timelineNavigator.getNavigatorName() +"_"+ APP_TITLE'***, where there is a '***timelineNavigator'*** variable, which calls the ***getNavigatorName***() method, followed by a dashed line and then the '***APP_TITLE'*** variable. Our tool effectively extracted the required information from this example into one table. However, it cannot understand the composite value of this parameter (which requires the method call to be resolved and then concatenated to the variable's value); so this needs to be

investigated further to gather the rest of the information. Unfortunately, our tool does not cover these processes, as it deals with syntax analysis without the semantic understanding. Thus, this tool finds it hard to identify the name of the widget declared in this way.

During this experiment, it was found that the code style can be considered as a critical element for the GUI analysis tool. The "Digital Parrot" application, for example, has a different code style than other examples described in chapter 4. This application is written by using one of the common object-oriented design patterns, namely, the abstract factory pattern.

In general, a design pattern is proposed as a best solution that is applied successfully in various environments to solve a problem that occurs in a specific set of situations. Alexander et al. describes a pattern as a recurring solution to a common problem in a given context and system of forces [Alexander77].

Design patterns are categorized into different groups: creational patterns, structural patterns, and behavioral patterns; all of these are described in detail in [Kuchana04]. One of the most common and well-used creational patterns is the factory pattern, which is a simple technique used to produce objects in a class. Factory patterns are designed to implement the concept of factories and deal with the problem of producing products without specifying the exact class of object that will be constructed. For example, a widget library needs to create multiple related widgets - buttons, labels, frames, text fields, etc., the abstract factory is an appropriate solution for this.

The abstract factory also offers a way to encapsulate a group of individual factories for creating families of related objects without specifying the actual concrete classes [Freeman04]. The generic class for the abstract factory is shown in the Figure 5.20.

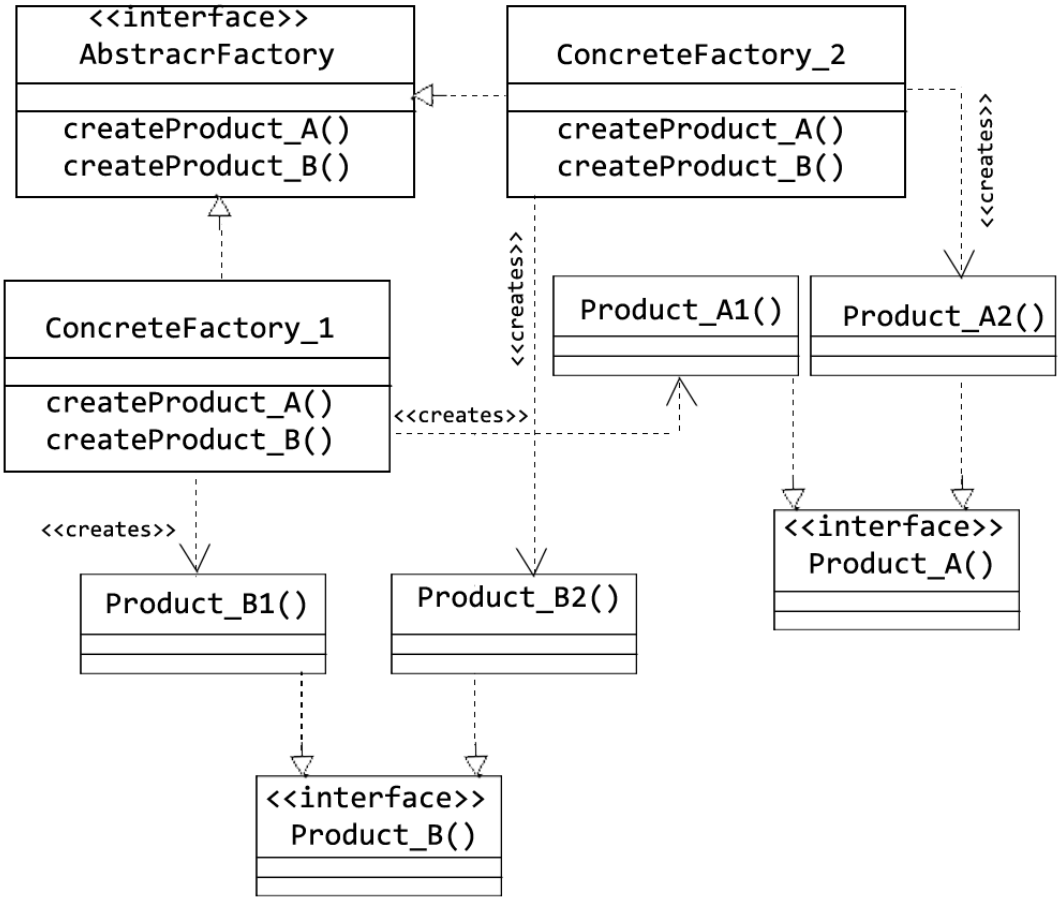


Figure 5.20. The class diagram for the abstract factory pattern

The typical structure of the Abstract Factory is as follows:

- Abstract Factory – declares an abstract interface for creating objects in a product family;
- Concrete Factory - produces concrete product objects in a product family by implementing the interfaces provided by the abstract factory class;
- Abstract Product – declares an interface for a type of product;

- Concrete Product – declares a concrete product of a product by implementing the abstract Product interface, to be generated by the corresponding concrete factory; and
- Client – uses the interfaces declared by abstract factory and abstract product classes to create the objects without needing to know which concrete class is actually instantiated.

The following example, which is an application to create the GUI elements of different named Buttons will illustrate this . For simplicity, two different buttons are considered: Ok_Button and Cancel_Button

Following is the ***Button*** class interface, which will be returned as the final end product from the factories.

```
public interface Button {
  public String getButton();
}
```

Next is the abstract ***GUIFactory*** class, which all factories will return.

```
public abstract class GUIFactory {
public abstract Button createButton();
}
```

Finally, two concrete factory classes– **OK_ButtonFactory** and ***Cancel_ButtonFactory*** - are defined as concrete subclasses of the ***GUIFactory***

```
public class OK_ButtonFactory extends GUIFactory {
 public Button createButton(){
  return new OK_Button("OK");
  }
```

```
public class Cancel_ButtonFactory extends GUIFactory {

 public Button createButton(){

  return new Cancel_Button("Cancel");

  }
```

The following classes show two concrete product classes- ***OK_Button*** and ***Cancel_Button***-, which implement the ***Button*** class.

```
public class OK_Button implements Button{

    public String name;

    public Button(String name) {
      this.name = name;
     }

    public String getname () {
         return this.name;
   }
}
```

```
public class Cancel_Button implements Button{

    public String name;

    public Cancel_Button (String name) {
      this.name = name;
     }

    public String getname () {
         return this.name;
   }
}
```

Finally, the client Element class invokes the ***getGUIelement(String GUIButton)*** method on the ***GUIFactory*** class. This method creates a proper factory product and returns it as an object of ***GUIFactory*** type. The test of this abstract factory design pattern is presented on this client class.

```
public class Element {

    private GUIFactory GUI;

    public GUIFactory getGUIelement(String GUIButton) {
      if (GUIButton.equals("OK_Button"))
          GUI = new OK_ButtonFactory();
          else if(GUITButtone.equals("Cancel_Button"))
             GUI = GUIButton new Cancel_ButtonFactory();

     return GUI;
        }

   public static void main(String[] args) {
     Element name = new Element();
     GUIFactory GUIElement = name. getGUIelement ("Cancel_Button");
     System.out.println("This is a "+
GUIElement.createButton().getname()+ "button");
      }
}
```

In this example, the correlations are:

- AbstractFactory => GUIFactory
- ConcreteFactory => OK_ButtonFactory and Cancel_ButtonFactory
- AbstractProduct => Button
- ConcreteProduct => OK_Button and Cancel_Button
- Client => Element

The output of this example will be "*This is a Ok button*" or "*This is a Cancel button*". This is dependent on the factory used. In this example, there no mention of the type of ***GUIFactory*** that occurs or the kind of ***Button*** that is produced by the factory.

Programming GUIs using this style produces very different code from the previous examples, which do not use patterns. Thus, understanding such patterns and their ways of creating widgets is an important step in discovering all possible ways to capture these widgets and their behaviors. However, it also presents the problem of needing to know whether a pattern-based programming approach has been used and

which patterns are involved etc. It is beyond the scope of this work to examine this further, but this will be discussed in the future research section.

5.5. Summary

This chapter described in detail all the initial investigations to create the PModels and PIMs in this study. All these attempts aim to capture the required information from existing code to produce correct and full PModels. This chapter started with investigating whether clone detection can help in our analysis process to reduce the complexity of the code. Three methods are used to achieve this: using the clone fragments; removing all the clones fragments and keeping just one copy of each set; and finally, removing all the clone fragments by using a specific heuristic procedure to keep the information that is responsible for creating widgets or their behaviors. During this experiment, it was found that there is a slight reduction of the information in the source code and PDGs, but either the clone detection does not reduce the PDG complexity or removes too much information. This means that clone detection cannot help during the interactive reverse engineering processes.

This chapter also described the static analysis method by using the PDGs to extract the widgets and their behaviors. In this experiment, the "*Finder dependencies*" tool was used to generate the PDGs automatically for the interactive applications. We found that we can extract the widgets and the behavior types of these widgets from the graph, but it is hard to identify the widget names, behavior names, and the link between these widgets to create the structure of the models. Thus, to solve this problem, we suggested using one of the dynamic analysis tools to investigate whether combining the final information from two approaches could help generate the full models. However, this attempt was unsuccessful in achieving this, due to missing information that is important to link extracted models from both dynamic and static methods to create the final PModels.

This chapter ended by describing the static analysis of source code to extract all the GUI elements and their behaviors to create PModels and PIMs. In this experiment, we extracted the widgets by detecting the widget data types and identified the relation between these widgets via detecting all the embedding widget actions in the code; this helps to create the structure for the models. Furthermore, from the code inside the event handlers of the widgets, we can discover all possible behaviors for each widget. Finally, based on the algorithm described in this experiment, we were able to implement an automatic tool to generate the desired models. At the end, we found that this experiment solved the previous experiments' problems in terms of identifying the widget names, behavior names and types, and widget hierarchies. However, the problem with this attempt is that it does not work well with all interactive applications. This is because this analysis is able to capture all the symbols (e.g. variables, classes and methods) in code, but it does not capture all the interaction between these symbols. This means that the experiment focuses only on syntax analysis and ignores the semantic side, which is important to understand the code, whatever its style, and helps to track and check the meaning of the expressions in the code to effectively extract all the widget information and their behaviors. This experiment cannot capture this meaning because our analysis is based on reading the code one time and extracting all the required information in one table, which in turn is used to produce the PModels.

To solve this problem and overcome code style problems, we suggest capturing the symbols and their information from the code into a symbol table, and parsing the code to build a parser tree, which enables us to visit element by element and track each element in this tree more than once, depending on the need to check the actual

meaning of the statements and the interactions of these elements in the code. This is described in the next chapter.

Chapter VI

A Technique for Reverse Engineering GUIs

In Chapter 5, we showed the different attempts and experiments of reverse engineering analysis to build the formal models, and described our attempts to operate these experiments automatically to generate the PModels. Through these experiments, we found that it is difficult to build the full PModels due to missing information in PDGs, and even after combination with the dynamic method using the GUI Ripper tool, this loss of information stands as an obstacle to generating the full and correct models. Moreover, we found that GUI elements and their behaviors can be identified from direct static code analysis. However, the PModels cannot be obtained in all the GUI applications; this is because this approach deals only with syntax analysis, where we are able to extract all the GUI components without understanding the expressions between these extracted elements.

Actually, in all previous experiments we have deliberately focused on the programmatic side rather than making theory suggestions and solutions to extract the models. This technique in the investigation and exploration led to an investigation of all the possible problems in each experience practically. Particularly in the last experiment, which relied on extracting the models from the static source code, the theoretical part of the analysis helped us to understand the structure of the Java program, but when we tried to automate a tool for this process - by separating each statement in the code into tokens to be used to identify the desired information on the basis of the extraction algorithm - we found that this tool did not work effectively with all GUI applications. Therefore, we have tried to identify the problems so that

we can investigate how to solve them. Finally, we found that a convenient solution for all GUI applications, whatever their code style, was to track the elements in a program and understand their expression and the relation between them. The solution can be made easier by traversing the parser tree.

Thus, we now discuss one of the advanced approaches of static analysis methods. The general strategy is to parse an interactive system code into a parser tree and then traverse that tree to collect all the desired information to build both the PModel (the presentation model) and PIM (the presentation interaction model). This chapter presents our technique for static reverse engineering an interactive system to obtain the required models from the parser tree. The ANTLR tool is used for automatically generating the parser trees of an interactive system and for its tree-walking mechanisms. Moreover, chapter 7 examines this approach with our case study – "BMI Calculator" application - and illustrates how to extract the program entities and use the walker method to extract the information required to produce the PModels and PIMs. It also handles some associated issues with a complex/advanced example (the "Digital Parrot" application) in chapter 8.

6.1 The approach

Our approach explained in this section assists in identifying a GUI from the legacy code. The goal is to detect the GUI's elements and all the related interaction and functionality behaviors of these elements in the user interface.

In order to attain our required models from a Java/Swing program, a parser is used to obtain the parser tree from source code. The straightforward approach is to extract all the symbol definitions (e.g. a variable/function/type declarations, etc.) and their relevant information from the code into tables, and then traverse the tree, and track and check each identifier to determine its type, and how this identifier interacts in expressions. The tracking and traversing of the parser tree can be achieved by using tree walker methods. This helps to capture all information needed in terms of the GUI's components, their types and all the possible behaviors (e.g. interaction, and function behaviors that occur within the systems) in the UI, where all this captured data is represented in the form of both PModels and PIMs. Thus, our reverse engineering approach uses the tree walker methods and focuses only on the nodes of interest to extract the desired information from the tree. Figure 6.1 illustrates our approach.

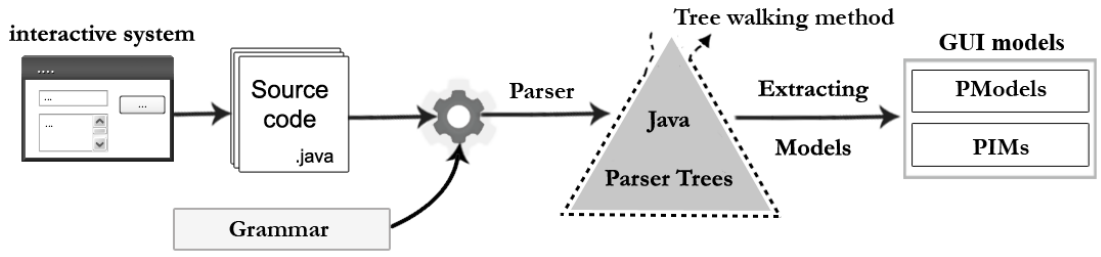


Figure 6.1. Presents the process of reverse engineering.

6.2 Tool

In order to achieve our goal of an approach for reverse engineering of GUI source code, ANTLR [9] is used in this paper for generating the AST of interactive Java source code. ANTLR tool is a parser generator framework, the first being developed in 1989 by Parr. It is written in Java and is under active development. We used this tool for many reasons. Firstly, this tool is free and is effective software. Secondly, it can be used to read and translate the grammar rules of a variety of programing languages (e.g. C, C#, Java, JavaScript, Python, etc.) to generate lexers, parsers, and tree parsers. Parsers can automatically generate abstract syntax trees (AST). There are a number of recent and stable versions of ANTLR. However, in this paper, ANTLR v4.0 is used, mainly because of the new additional features. The most remarkable feature in this version is that ANTLR v4.0 provides support for automatically generating two tree-walking mechanisms in its runtime library: (1) Parser-tree listeners that can be used to listen for "enter" and "exit" events of each rule; and (2) Parser-tree visitors that can help in building tree walkers that visit or rewrite the trees. From using any either of these tree walkers, we are able to traverse the parser tree and collect all the desired information to produce our required models.

Generating the parser tree for any phrases needs specific grammar rules. To clarify further, Figure 6.2 below shows a simple grammar that identifies expressions like: Z= 30.

```
grammar Example:   // Define a grammar name as Example
stat : ID '=' expr '.' ;   // match an assignment statement like “Z=30” or"X= 100;"
expr: INT ;
ID : [a-zA-Z]+ ;          // match identifiers
INT : [0-9]+ ;            // match integers
```

Figure 6.2. Segment of grammar symbols

[9] http://www.antlr.org

The rule as stated above can produce the parser tree, as shown in Figure 6.3 below.

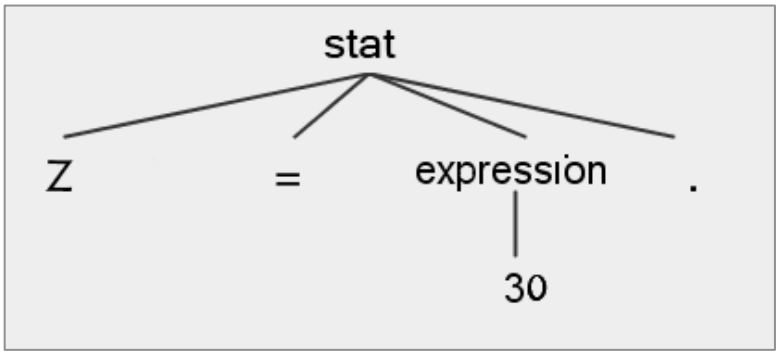


Figure 6.3. Parser tree output from simple grammar

Internal parser tree nodes refer to the rule applications, and leaf nodes refer to the token matches based on these rules.

In our study, the grammar of Java.g4[10] is implemented to create the parser tree of the systems written in Java.

Tree-walking techniques

Having created an appropriate parser tree of Java source code with all the necessary information, we can start deriving the information required. The advantage of the ANTLR V.4 tool is that it generates two types of tree walkers, listeners and visitors. Our goal in this section is to give an adequate concept to understand exactly how those tree-walking mechanisms can be used.

As a convenience, let us look at the following Java/Swing code fragment as an example:

```
JButton quitButton = new JButton("Quit");
```

Parsing this code snippet above, the following fragment of the parse tree is obtained (Figure 6.4).

[10] https://github.com/antlr/grammars-v4/blob/master/java/Java.g4

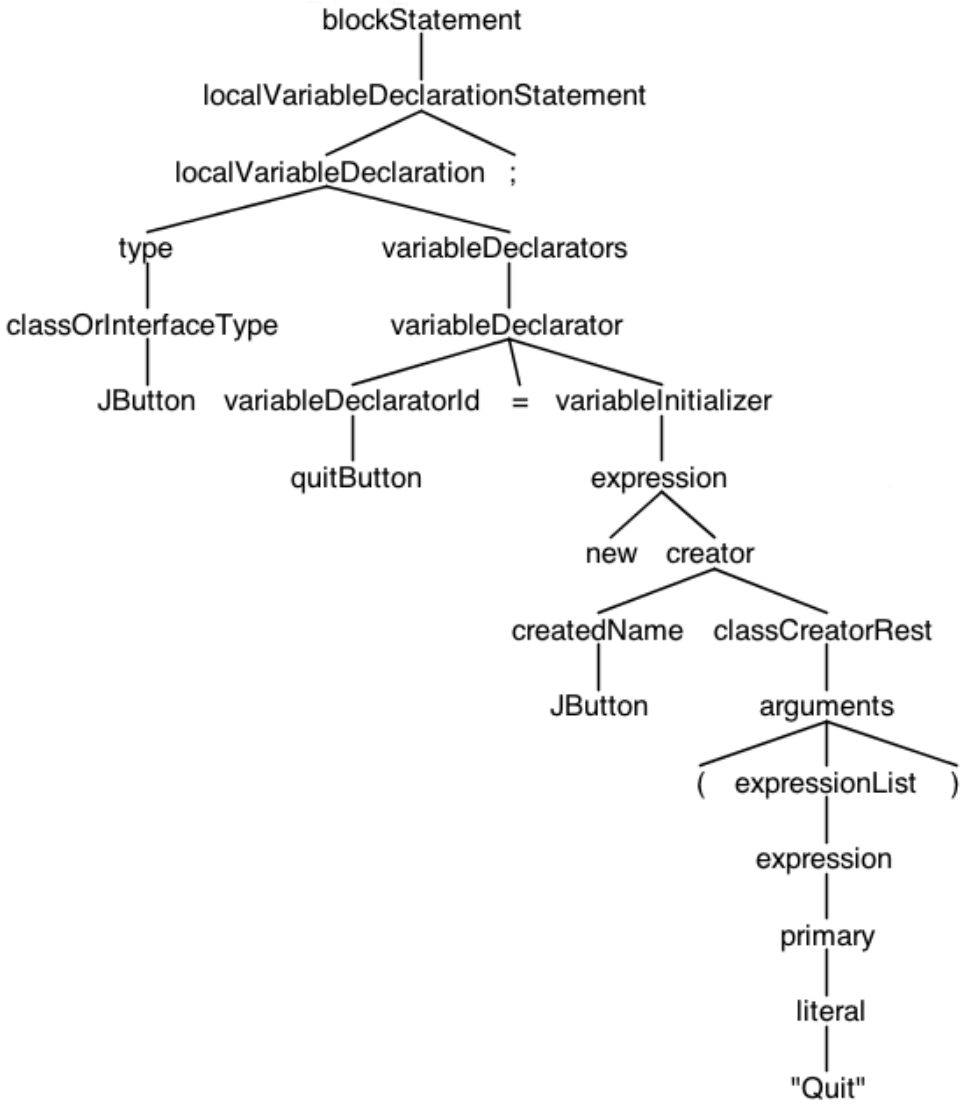


Figure 6.4. Represents part of the parse tree for “BMI Calculator” program.

From the Java grammar/parser tree of this particular fragment, we are able to use the walker methods on only those parts that correspond to the particular statement and ignore all of the other non-interesting nodes. The parser tree for the specific statement, in turn, has too much non-interesting information; we also need to focus on the certain nodes that we want to read in the tree. In the figure above, for example, we only need to walk through three of the statement rules to read and extract the desired information.

Parr stated that listener and visitor walkers are different in that listener methods are not responsible for exact calling ways to walk their offspring, while visitors must clearly instigate visits to child nodes to maintain tree traversal [Parr12]. In this research we will focus only on the Parser-tree listener walker method.

Parser-Tree Listeners

By using the parser-tree listener walker technique, we will able to "*enter*" and "*exit*" nodes of specific parser rules that match tokens for '*JButton'* constructor assignment in the fragment above.

ANTLR automatically creates a listener interface with the relevant methods for each rule. Some of the relevant methods from the generated listener interface are shown in Figure 6.5 below .

```
public interface JavaListener extends ParseTreeListener {

  void enterInnerCreator(JavaParser.InnerCreatorContext ctx);
 void exitInnerCreator(JavaParser.InnerCreatorContext ctx);
 void enterExpressionList(JavaParser.ExpressionListContext ctx);
 void exitExpressionList(JavaParser.ExpressionListContext ctx);
 void enterVariableDeclarators(JavaParser.VariableDeclaratorsContext ctx);
...

}
```

Figure 6.5. Segment of generating listener methods.

ANTLR also generates a default listener implementation called *JavaBaseListener* that identifies all the enter and exit methods for rule assignments. In this case, we can create a class that should extend *JavaBaseListener*. To extract the required information, we override the procedure which triggers callbacks to a listener when seeing the beginning and end of the various statements and expressions. The tree diagram in Figure 6.6 shows the method call to the listener for the *localVariableDeclaration* rule.

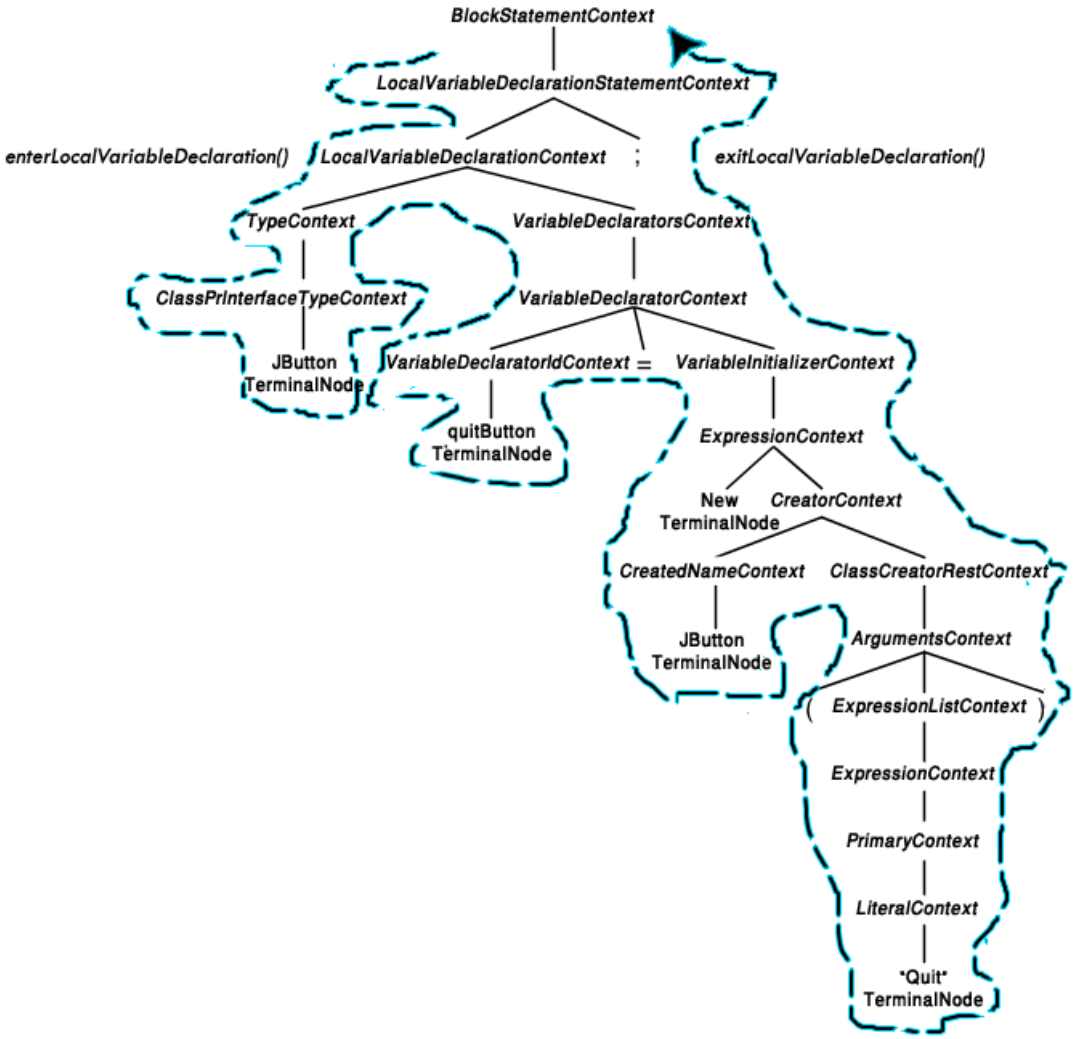


Figure 6.6. The parser tree walker that uses the enter and exit methods for *localVariableDeclaration* rule

All the listener method calls for the specific rules for the constructor assignment are shown in Figure 6.6. The thick dashed lines represent the parser tree walker. The walker will encounter the node for rule *localVariableDeclaration*, where the walker triggers *enterLocalVariableDeclaration*() and passes it to the *LocalVariableDeclarationContext* node; visits all the children of the *localVariableDeclaration* node, and then passes to exit by triggering *exitLocalVariableDeclaration*() method.

To extract the desired information from the '*JButton*' constructor assignment, we need to know the rules that are responsible for matching the variable name, its type and value, and then use their listener methods to produce their signatures. Below is the code that is able to extract the information required.

```
public class Test extends JavaBaseListener {
public static  String typeOfval,val, Value;
.
.

//extract the Variable's name
 public void
exitVariableDeclaratorId(JavaParser.VariableDeclaratorIdContext ctx)
    {
        val =ctx.Identifier().getText(); // String
       System.out.println("variable name:"+ val);
    }

//extract the Variable's type
  @Override
 public void
exitLocalVariableDeclaration(JavaParser.LocalVariableDeclarationContext
ctx) {
        typeOfval=ctx.type().getText();
        System.out.println("variable type:"+ typeOfval);

        }

//extract the contracture's parameter
@Override
 public void exitPrimary(JavaParser.PrimaryContext ctx) {
       if (ctx.literal() != null){
      Value= ctx.literal().getText();
      System.out.println("variable value:"+ Value);
     }
}
```

After the parser has been launched, the output of this example will be:

Variable type: JButton

Variable name: quitButton

Variable value: Quit

The following chapter will illustrate how to use the parser tree and the walker method to extract the information required to produce the PModels and PIMs.

Chapter VII

Deriving GUI models from source code via the parser tree

The previous chapter gave explanations of the ANTLR tool and its ability to generate the parser tree for any language based on the grammar and build tree walkers automatically that listen to and visit those trees. That means we are almost ready to analyze the statements of code by using walker tree methods to extract and track *symbol* definitions and then we are able to identify the GUI elements easily. "A *symbol* is just a name for a program entity like a variable or method" [Parr12]. Usually, a symbol table is used to track symbols in language applications. This table is commonly used as an important part in compiler and interpreter applications. To build *Presentation Models* (PModels) and *Presentation Interaction Models* (PIMs), it will be used to extract the GUI elements and all the desired information from the parser tree.

This chapter will describe how we define all the symbols (program entities) in the Java applications and sort these symbols into symbol tables and then detect the GUI components from these tables. This chapter also tries to understand the meaning of the statements in Java through its generated parser tree to derive the required information, such as relationship between the symbols (widget hierarchies), and their behaviors.

7.1 Symbol table and detecting the GUI elements

In general, when reading and writing a program, all the applications are usually found to have the same rules and structure to execute the code. To build a symbol table, the

general structure needs to be formalized and how the program entities are represented in any Java software need to be identified. The following Java code is an example.

```
public class MBI_calculater extends JFrame  {
    String W[]={"kilograms","pounds"};
    JButton quitButton;
    public void quit_fun() {
     ...  }}
```

The segment above, defines four *symbols* (program entities): class ***MBI_calculater***, function ***quit_fun***, and two variables ***W []*** and ***quitButton***. Frome ach one of those definition symbols needs to collect some information. The information needed for each of those definitions has at least the following properties:

- Name
- Symbol Category: indicates exactly what kind of the symbol, (e.g. a class, method, variable, and so on).
- Type (e.g. int, String, Button, etc.)
- Value
- Scope: a set of symbols such as a list of parameters for a function or the list of variables and functions in the global scope.

Language Implementation Patterns [Parr09] discusses symbol table management in more detail. [Parr12] details how to implement an appropriate symbol table using the listener walker method (generated automatically by using ANTLR v.4), where the tree walker triggers enter and exit events for particular nodes to identify and extract all the programme entities and their properties, based on the grammar rules.

This research has used the symbol table source code [11] from Chapter 6 of *Language Implementation Patterns* [Parr09]. Their *symbol table* implementation is based on two basic processes: (1) defining all symbols in their associated scope; and then (2) resolving the *symbol* by using the resolve() method. From the output of this

[11] http://pragprog.com/book/tpdsl/language-implementation-patterns

implementation, we can extract our models. A nested scope for class is used to deal with class inheritance in object-oriented languages (OOLs). Figure 7.1 shows the relationship between *symbol table* objects in an OOL.

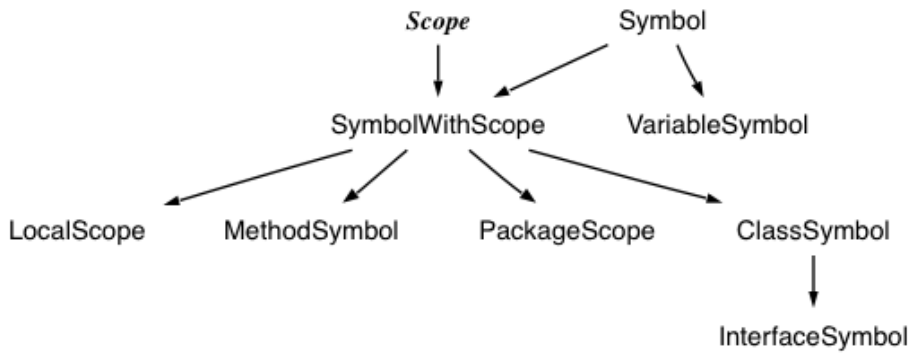


Figure 7.1. The class hierarchy for a *symbol table* (Adapted from [Parr09])

This can help us to handle all interactive applications, even complex applications that are written using the object-oriented design patterns (see the Parr09 for more details), most of which are based on class inheritance. Our current analysis in this area will be used to extract the PModels and PIMs. Future research can complete and improve this code to be used for extracting the models automatically.

Some modifications to the basic code have been made for convenience of use with Java grammar rules that are used in this study. We also tried to collect the values of some symbols, such as the items in a specific array contents or initial values of some symbols. In [Parr12], for example, each *symbol* is implemented with a separate class, holding the name and its properties, which are type and its scope. We added the value as one of the *symbol* properties. The reimplementation for the *Symbol* superclass can be:

```
public class Symbol {
    String name;    // all symbols at least have a name
    String type;    // Symbols have type
    String value;   // some variable symbols may have a initial value
    Scope scope;    // to know the scope for the symbols.
```

Based on this change, the *VariableSymbol* class then will be:

```
public class VariableSymbol extends Symbol{
    public VariableSymbol(String name, String type){
       super(name, type);
     }
     public VariableSymbol(String name, String type, String value){
        super(name, type, value);
     }
    public String toString(){
        return super.toString();
     }
}
```

Finally, after building and testing the parser tree, the desired output for "BMI Calculator" program will be as shown in Figure 7.2.

```
Define class :MBI_calculater its superclass: JFrame
     Defin field:weightLabel, of type: JLabel Values : []
     Defin field:weightLabel1, of type: JLabel Values : []
     Defin field:HeightLabel, of type: JLabel Values : []
     Defin field:HeightLabel1, of type: JLabel Values : []
     Defin field:jPanel1, of type: JPanel Values : []
     Defin field:jPanel2, of type: JPanel Values : []
     Defin field:weightTF, of type: JTextField Values : []
     Defin field:HeightTF, of type: JTextField Values : []
     Defin field:str2, of type: String Values : []
     Defin field:str, of type: String Values : []
     Defin field:res, of type: tINVALID Values : []
     Defin field:flage, of type: tINT Values : [0]
     Defin field:ImagIc, of type: String Values : ["Images/a0.png", "Images/a1.png", "Images/a2.png", ...]
     Defin field:text, of type: String Values : [" ", "underweight", "normal weight", " overweight", "obese"]
     Defin field:imageIcon, of type: ImageIcon Values : []
     Defin field:ResLabel, of type: JLabel Values : [imageIcon]
     Defin field:Titel, of type: JLabel Values : []
     Defin field:frame2, of type: JFrame Values : ["Result"]
     Defin field:W, of type: String Values : ["kilograms", "pounds"]
     Defin field:WCB, of type: JComboBox Values : [W]
     Define Function :itemStateChanged
     Function <locals.itemStateChanged:tVOID>:[<ie:ItemEvent>]
     Defin field:H, of type: String Values : [ "inches", "centimeters"]
     Defin field:HCB, of type: JComboBox Values : [H]
     Define Function :itemStateChanged
     Function <locals.itemStateChanged:tVOID:[iee:ItemEvent>]
     Defin field:ClcBtn, of type: JButton Values : ["Calculate"]
     Define Function :actionPerformed
     Function <locals.actionPerformed:tVOID>:[e:ActionEvent]
     Defin field:closeButton, of type: JButton Values : ["Close"]
     Define Function :actionPerformed
     Function <locals.actionPerformed:tVOID>:[event:ActionEvent>]
     Defin field:ClButton, of type: JButton Values : ["Clear"]
     Define Function :actionPerformed
     Function <locals.actionPerformed:tVOID>:[event:ActionEvent>]
     Defin field:quitButton, of type: JButton Values : ["Quit"]
     Define Function :actionPerformed
     Function <locals.actionPerformed:tVOID>:[event:ActionEvent>]
     Constructor MBI_calculater.MBI_calculater:[]
     Define Function :quit_fun
     Function <MBI_calculater.quit_fun:tVOID>]
Class MBI_calculater:{weightLabel, weightLabel1, HeightLabel, HeightLabel1,
 jPanel1, jPanel2, weightTF, HeightTF, str2, str, res, flage, ImagIc, text,
imageIcon, ResLabel, Titel, MBI_calculater, quit_fun}
...
...

[@43,484:494='weightLabel',<100>,21:19]cymbol_name: weightLabel,Variable  <MBI_calculater.weightLabel:JLabel:[]>
[@47,516:527='weightLabel1',<100>,22:19]cymbol_name: weightLabel1,Variable  <MBI_calculater.weightLabel1:JLabel:[]>
[@51,549:559='HeightLabel',<100>,23:19]cymbol_name: HeightLabel,Variable  <MBI_calculater.HeightLabel:JLabel:[]>
[@55,581:592='HeightLabel1',<100>,24:19]cymbol_name: HeightLabel1,Variable  <MBI_calculater.HeightLabel1:JLabel:[]>
...
[@568,4074:4074='H',<100>,104:25]cymbol_name: H, Variable  <locals.H:String:["inches", "centimeters"]>
...
...
```

Figure 7.2. The partial *symbol* table output for "BMI Calculator" program

The output in Figure 7.2 above is divided into two groups: the first represents the result after defining the symbols in their scope, and the second represents resolving those symbols to be used for searching and verifying those defining symbols.

From the output above all the widgets in our GUI example are easily identifiable. Table 7.1 below shows a list of extracted elements.

Widget Type	Variable	Value
JLabel	weightLabel	-
JLabel	weightLabel1	-
JLabel	HeightLabel	-
JLabel	HeightLabel1	-
JTextField	weightTF	-
JTextField	HeightTF	-
JComboBox	WCB	W, which =[Kilogram, pound]
JComboBox	HCB	H, which =[inches, centimetres]
JButton	ClcBtn	-
JButton	ClButton	-
JButton	quitButton	-
JButton	closeButton	-
JLabel	Titel	-
JPanel	JPanel2	-
JPanel	JPanel1	-

Table 7.1 Elements extracted from implementation of symbol table.

At this point, all the symbols from code have been extracted and stored in a symbol table. The next section describes how to derive the information about the hierarchies, behaviors of the widgets and other information in interactive systems.

7.2 Collecting information about the symbols

Java code contains a number of different statements. The previous section has identified all the symbols and grouped them based on their kind, whether they are class, function or variable. However, the statements in code can be a constructor of a symbol, computing mathematical operations, calling methods etc. These statements need to be interpreted to identify and capture their meaning to extract the required information to build our models.

Understanding the meaning of the expressions in the code can be captured from the structure/syntax of a statement and the specific symbols used in this sentence [Parr12]. For example, there are many methods for importing from Java libraries that can connect with a particular symbol name to do certain tasks, such as, from the Swing library, *add* (), *setTitle*(), *setVisible*() and so on. Each method must end with two brackets "()" that may contain one or more parameters or none. Typically, the syntax to write the call of these methods is to write the variable followed by method name, separated by a ".", such as:

ResLabel . setText (" ");

To resolve an expression such as the phrase above, we resolve " ***ResLabel*** " and then look up "*setText* " within the static method that belongs to Swing Libraries. In this case, we can create this library as an "*array*" or "*switch*", that contains some key words required by static methods, such as *setTitle()", "add()", "setText()", "setVisible()","addActionListener()"*, etc. From those key words we can identify the information we need to extract based on the method's tasks.

In other situations, the *symbol* can call a method defined in another class in the program. This can happen when a class inherits from another class. In this case, the

method in the current class's scope needs to be researched. If it cannot be discovered the enclosing class's scope need to be investigated. Chapter VII discusses this in more detail.

Identifying the meaning of the statements is an important step to detect the information about widgets. The desired information can be divided into two groups: (1) widget embedding, and labelling; and (2) widget behaviors. Section 5.4 described how to detect those kinds of information from source code in detail. Similarly, extracting the information about the widgets and their behaviors can be done more easily by tracking the parser tree. Although, we were able to extract the models successfully for "BMI Calculator" examples earlier in Section 5.4, there were problems in some other interactive applications. This section also uses this example to simplify and clarify our analysis process. Later in chapter 8, we will show how tracking the parser tree can solve the problems that were defined in the previous chapter.

Detecting widget embedding and other relative information

The widget hierarchies help to detect the relation between each widget with others in any GUI system. Identifying this kind of relation is very important to create the final structure of the PModels: where, each PModel represents a window/frame in the application and any widget inside this window will be defined in the model as a list of triples (a widget's name, category, (set of behaviors)). The widget name represents a label/title of the element in GUIs, and category describes its type whether it is an *actionControl*, *Entry*, or so on. Widget behaviors describe the behavior of the action that occurs when a widget is clicked on; these behaviors can be an I_behavior, S_behavior or both. (See 2.2 for more details about the models).

Thus, to generate the PModels, we need to identify the widget embedding to build the model's structure, and recognize widget labels, and behaviors to describe the widgets in the models. The widget category is not needed because once the symbols with their data type (described in the previous section, 6.1) have been extracted; the category of the widget can be determined easily from its data type.

To collect the required information from the parser tree, the "*BMI Calculator*" application is divided into slices and each slice is explained separately. The following Java/Swing code fragment is used as an example.

```
frame2.setTitle ("BMI Result");
```

Figure 7.3 represents the output parse tree for (*frame2.setTitle ("BMI Result")* fragment.

Figure 7.3. Output parser tree for (frame2.setTitle ("BMI Result") fragment.

To extract the desired information by using the listener walker tree, we need to identify the rules that are responsible for matching ".", to extract the left and right sub-tree's children of the "." node, where the left often represents a symbol/widget name, and the right represents a method name. The values of right and left are retained in memory for later use. To derive the parameter of the "*setTitle*" method, we need to call the exit method of the rule that matches token with "(", and then grab the right side, which represents an expression of method call phrase. In this case, we need to return the values from memory that contains the value of the left and right child of "." node. The second child of "(" node will be the parameter of the methods, which is, in this case, ***"BMI Result***".

To attain this, *exitExpression*() and *exitPrimary*() methods are used. Figure 7.4 clarifyies this. Dashed lines represent tree walkers of the *DotExpressionContext* and *PrimaryContext* nodes that can be called their enter and exit methods to check and extract the required information.

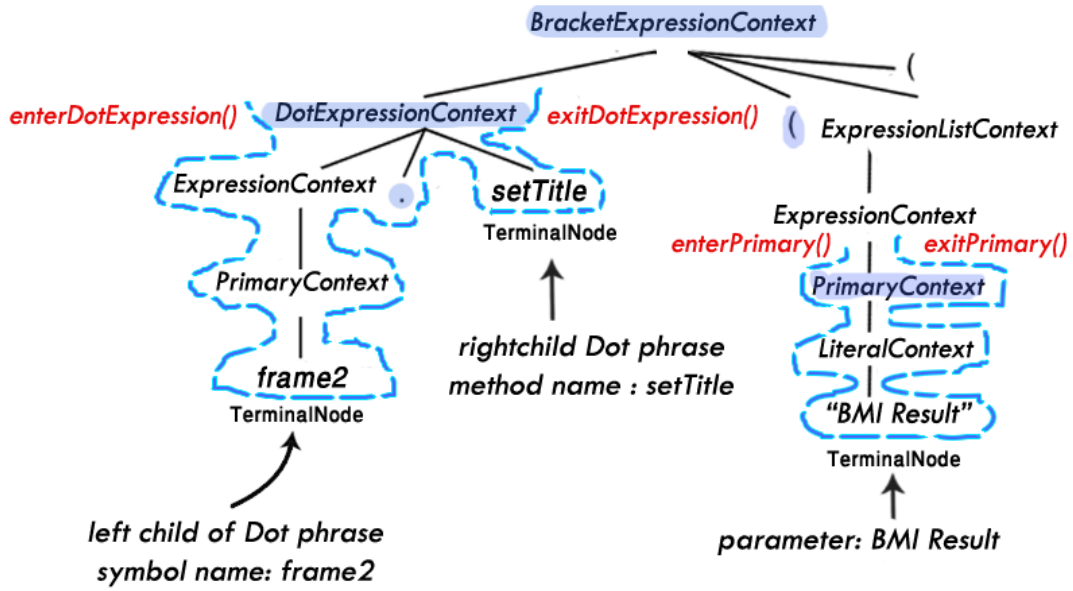


Figure 7.4. Tracking and checking the element in sub-children of DotExpressionContext and PrimaryContext nodes.

The code below shows how to identify the *symbol* and its method calls.

```
public void exitDotExpression (JavaParser.DotExpressionContext ctx)
  {
    if (ctx.getChild(1).getText().equals("."))
    {
      System.out.println("left: " + ctx.getChild(0).getText());
      System.out.println("right: " + ctx.getChild(2).getText());
    }
  }

public void exitPrimary(JavaParser.PrimaryContext ctx) {
       if (ctx.literal() != null){
        System.out.println("parameter ="+ctx.literal().getText());
      }
}
```

Based on this code, the output for *frame2.setTitle ("BMI Result")* statement, will be:

```
left: frame2
right: setTitle
parameter: "BMI Result"
```

Thus, the *symbol* "***frame2***" is type '*JF*rame'**,** and by using the "*setTitle*" method, we can say that the title of this *symbol* in the GUI is *"**BMI Result**"*.

Following this way and using the same code, it becomes easy to extract all the methods and the widgets that call them. Based on the method purpose, we can determine the desired information that we need to build our models. We can adjust the *exitDotExpression*() function in the code above, by using a "*switch*" to distinguish between the methods, and give the meaning of the statement to extract the information required. For example, during the tracking if we found "*add* ()" method, we can easily detect widget embedding, where the widget that calls this method will be the parent and the parameter represents the child widgets. The following code shows the reimplementation and the technique used to extract the embedding of widgets.

```
// "Memory" for the embedding; child/parent value pairs go here */

Map<Symbol, Symbol> embedding_memory = new HashMap<Symbol, Symbol >();
String P=; // to use for temporary keeping of the parameter value

public void enterPrimary(JavaParser.PrimaryContext ctx) {
       if (ctx.literal() != null)
        P=+ctx.literal().getText();
}

// xx.method();
public void exitDotExpression (JavaParser.DotExpressionContext ctx)
  {
   String left = ctx.getChild(0).getText(); //must be a symbole name
   String right = ctx.getChild(2).getText(); //could be method name
   Symbol W = currentScope.resolve(left);  // Resolve the symbol

   switch (right)  ) { // name of method
      case “add" :  // add
      {
        embedding_memory.put(W, P);   // store it in the memory
       break;
     }
      case “setTitle” :
     {
      System.out.println("the title of frame "+W+” is:”+ P);

      break;
     }
     case 3 :
      .
      .
  }

}
```

According to this code, the following output was produced:

```
{<locals.HCB:JComboBox:[H]>=jPanel1, <MBI_calculater.weightLabel1:JLabel:[]>=jPanel1,
<MBI_calculater.jPanel2:JPanel:[]>=frame2,<MBI_calculater.Titel:JLabel:[]>=jPanel2,
<MBI_calculater.ResLabel:JLabel:[imageIcon]>=jPanel2,<MBI_calculater.HeightTF:JTextField:[]>=jPanel1,
<MBI_calculater.HeightLabel:JLabel:[]>=jPanel1,<locals.WCB:JComboBox:[W]>=jPanel1,
<locals.ClButton:JButton:["Clear"]>=jPanel1,<MBI_calculater.weightLabel:JLabel:[]>=jPanel1,
<MBI_calculater.HeightLabel1:JLabel:[]>=jPanel1,<locals.closeButton:JButton:["Close"]>=jPanel2,
<locals.quitButton:JButton:["Quit"]>=jPanel1,<MBI_calculater.weightTF:JTextField:[]>=jPanel1,
 <locals.ClcBtn:JButton:["Calculate"]>=jPanel}
```

In this way, we can define the title of the frame objects in the application and all the elements that belong to each one of these windows. Table 7.2 shows some information that can be extracted from the tree of the “BMI Calculator “ application.

Frame/window title:	< BMI_mainWin: MBI_calculater > = “BMI Calculator”. <Frame2 :JFrame> = "BMI Result”.
Relations between widgets (widget hierarchies):	<quitButton:JButton> in jPanel1 <weightLabel1:JLabel> in jPanel1 <Titel:JLabel> in jPanel2 <jPanel2:JPanel> on frame2 <weightLabel:JLabel> in <weightTF:JTextField> in jPanel1 <closeButton:JButton> in jPanel2 <HeightTF:JTextField> in jPanel1 <ClcBtn:JButton> in jPanel1 <ResLabel:JLabel> injPanel2 <WCB:JComboBox> in jPanel1 <HeightLabel:JLabel> in jPanel1 <HeightLabel1:JLabel> in jPanel1 <ClButton:JButton> in jPanel1 <HCB:JComboBox> in jPanel1 < jPanel1: jPanel> in BMI_mainWin

Table 7.2. Partial output for some objects of “BMI Calculator” program

The information in the table is sufficient to build a structural form for PModels.

```
MainWin _Win:
(weight_Label, display, (...))
    (weight_Label1, display, (...))
    (WeightEntry, Entry, (...))
    (WCB_Sel, Container, (...))
```

(Kilograms_Item, SvalSelector,(...))
(Pounds_Item, SvalSelector,(...))
(Height_Label, display, (...))
(Height_Label1, display, (...))
(HeightEntry, Entry, (...))
(HCB_Sel, Container, (...))
(Centimeteress_Item, SvalSelector,(...))
(Inches_Item, SvalSelector,())
(Caculate_Button, ActionControl, (...))
(Clear_Button, ActionControl, (...))
(Quit_Button, ActionControl, ((...))
BMI_Result _Winis
(Result _Label, display, (...))
(Close_Button, ActionControl, (...))

Detecting widget behaviors

This example shows how to identify some required information from the tree by using the listener walker technique. To detect the behaviors of a symbol/widget from the tree, the same algorithm used in our analysis in Section 5.4, will be used to detect the behavior types and names of each widget.

As a convenience, we create a button that closes frame/window, called *fram2*. The *addActionListener* method is used to register the Listener to the item *closeButton*, when clicking on this button; *actionPerformed* method is invoked to handle the event.

```
closeButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
       Close_fun();

     }
 });
```

The following tree diagram shows the parser tree of the segment statement above.

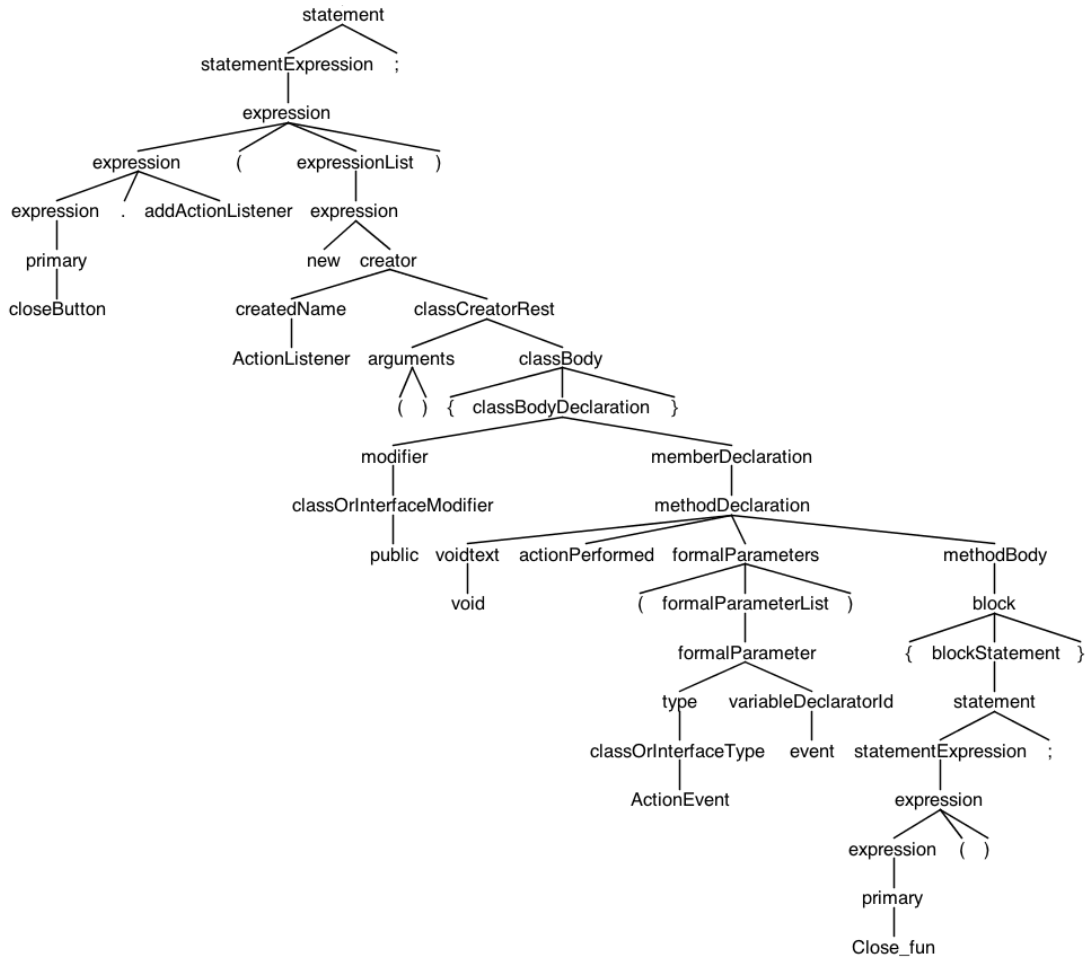


Figure 7.5. Output of parser tree

To extract the required information, the same steps as described in the previous example are followed. Only we will focus on a particular node in the parser tree and ignore the others; where we will call the exit method of the rule that matches the "." token, which is *exitExpression()*. The right side should be the symbol/widget name, so it needs to be resolved and kept in memory, and the left child represents a method call, so the purpose of using this method needs to be checked. If it is one of the registered method names, the event handler needs to be found to identify the behaviors (see Figure 7.6 for clarification).

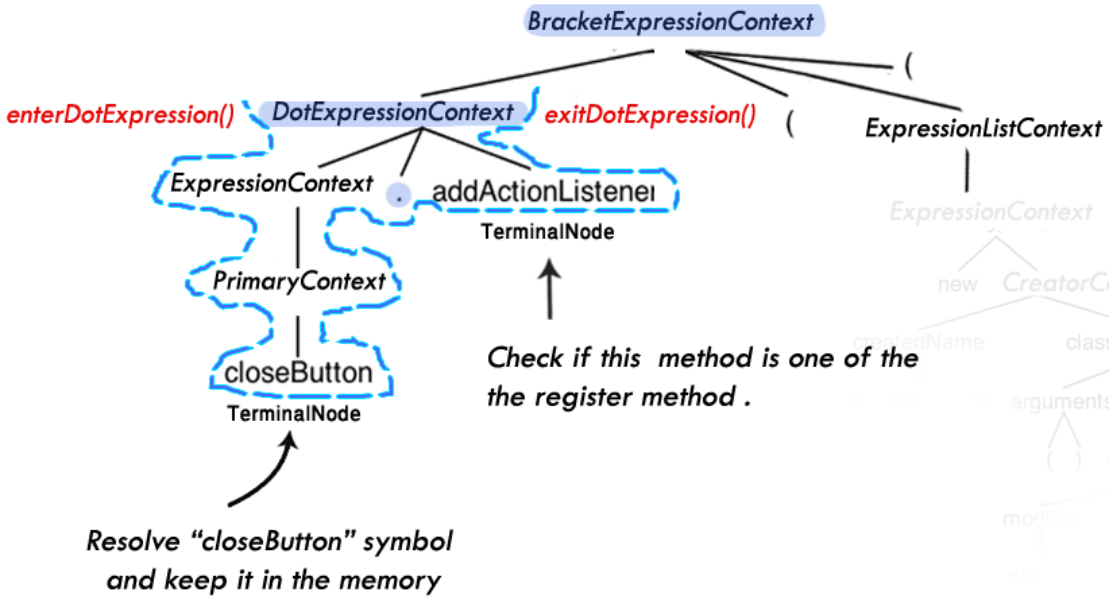


Figure 7.6. Tracking and checking the method call

In our example, the static method is "*addActionListener*"; this means that an event handler should be invoked in one of the next nodes in the right side of the rule that matches the "(" token. In our case, the event handler is "*actionPerformed"*. Thus, we need to check the behaviors in each node inside the event handler (see section 5.4, for more details on detecting the behaviors types and names of the widget).

Here, "***Close_fun*** ()" procedure is called inside "*actionPerformed"*. "***Close _fun*** ()" function does not belong to I_behaviors group, (listed in Table 5.6, in section 5.2), or S_behaviors group, (which represents other static methods that do not use any of the methods that belong to I_behaviors group). Thus, we need to resolve this function from the symbol table and then track and check all the nodes inside this function to realize whether the behavior is an I_behavior or S_behavior. The following fragment represents the code of the "***Close_fun***()":

```
public void Close_fun() {
      frame2.setVisible(false);
 }
```

Figure 7.7 shows the parser tree of the "***Close_fun***()" statements.

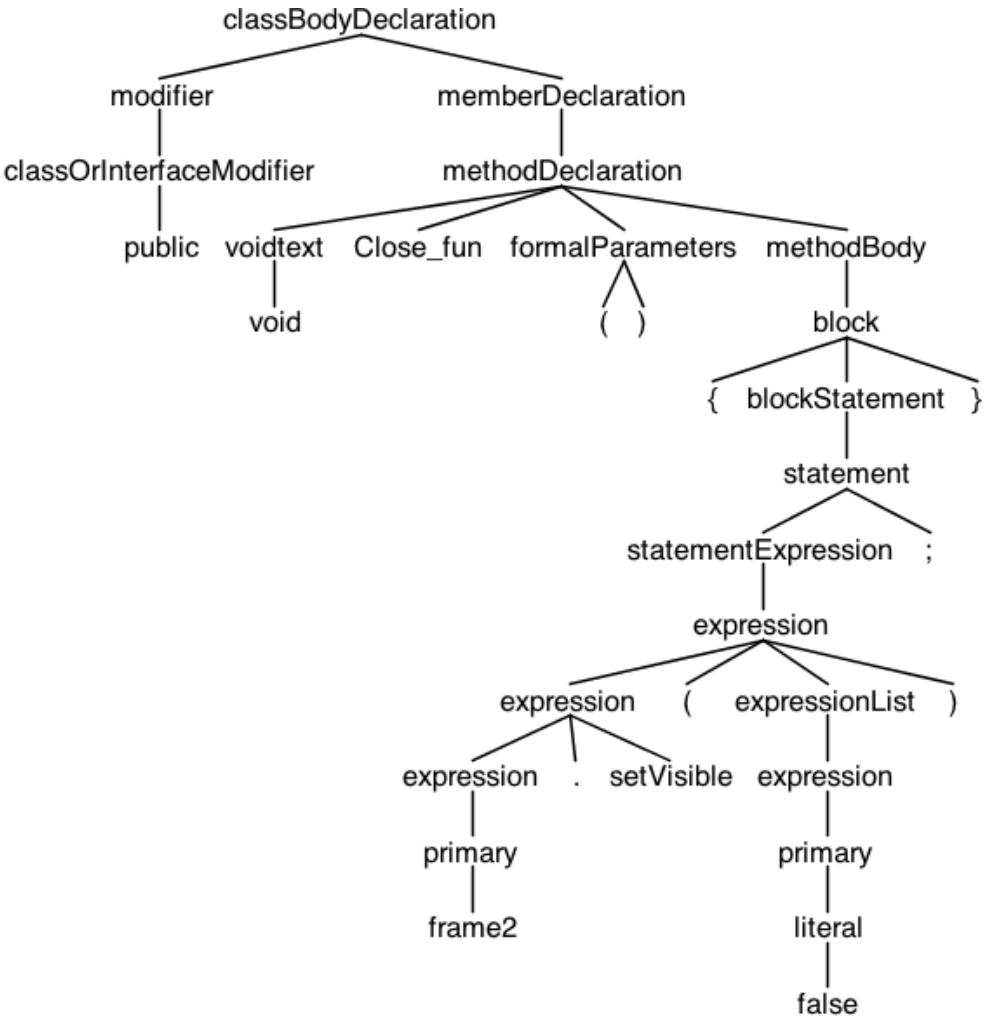


Figure 7.7 Parse tree for code of the Close_fun() .

From the tree diagram, (Figure 7.7) we track and search for the nodes that belong to any behavior groups in the function scope highlighted in green in the parser tree (see Figure 7.8).

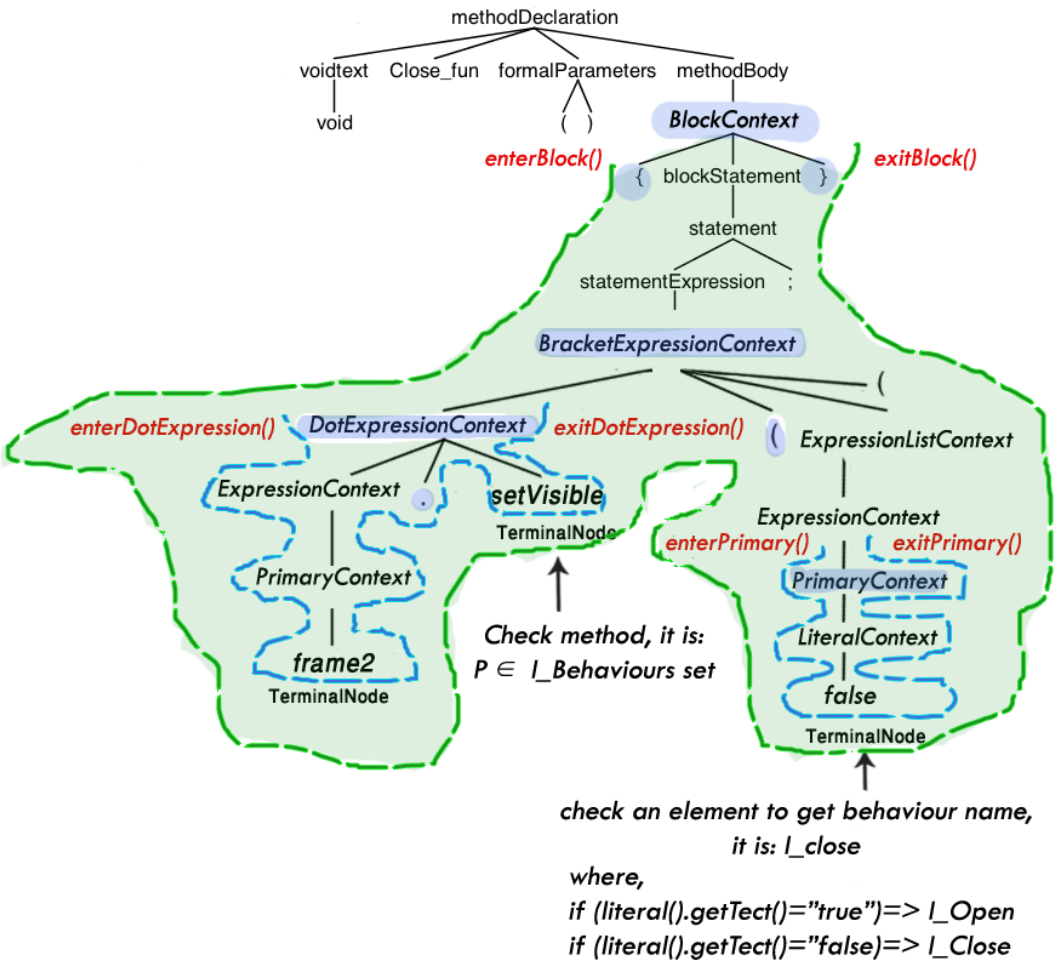


Figure 7.8. The function scope area, and the exit and enter listener methods used for particular nodes to extract the information.

In Figure 7.8, the green thick dashed lines represent the parse tree walker for the function scope area to collect all the behaviors inside the function, and the blue thick dashed lines represent the parse tree walker for the nodes from which we want to extract information. The walker will encounter the node for rule ***BlockContext***, where the walker triggers ***enterBlock***() and pass it to trigger ***DotExpressiont()*** method and visit all the children of the ***DotExpressionContext*** node to resolve a symbol and check a method call to detect the behavior type, and then pass to check the parameter to identify the name of the behavior , finally pass the tree to exit from the function scope by triggering ***exiBlock***() method.

We found one node matching "*setVisible*" method with a "*false*" parameter. The "*setVisible*" method belongs to I_behavior group, and the parameter "*false*" means when this function is called, "***frame2***" will close. In this case, the behavior is an I_behavior and its name is "*Close*". According to all this information, the PModels of this example will be as follows:

(Close_Button, ActionControl, (I_Close))

We also can build the PIM shown in in Figure 7.9.

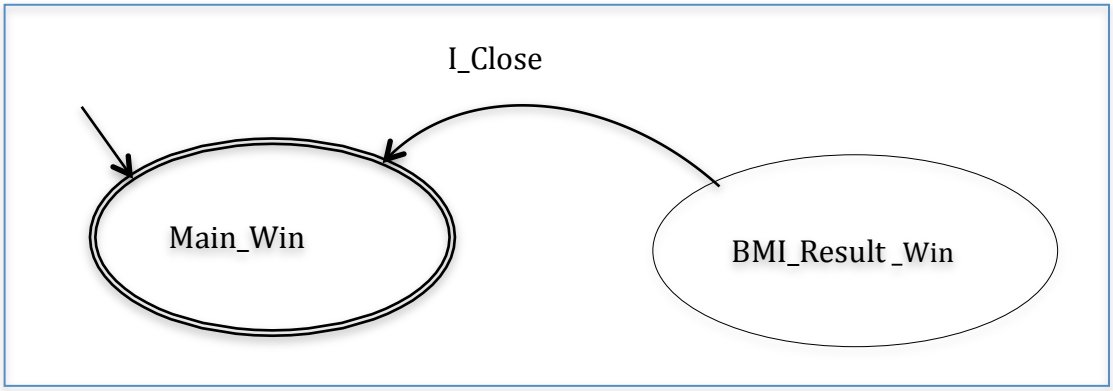


Figure 7.9. Interaction behavior for close button.

In the implementation, all the extracted behaviors in the event handler can be pushed into a stack, and at the exit of the node that matches the "}" statement rule, we can pop the stack to sort all the possible behaviors to connect the result of the behaviors with the symbol /widget stored in the beginning of this part of the tree in the memory.

In this way, we can detect all the behaviors of each widget. Thus, the final PModels of the "BMI Calculator" application will be as shown as follows.

```
MbiCalculater is MainWin : BMI_Result _Win
        MainWin is
           (WeightEntry, Entry, ())
           (WeightEntry, SvalueResponder, (S_clear))
           (WeightSel, Container, ())
             (KilogramsItem, SvalSelector,())
             (PoundsItem, SvalSelector,())
           (HeightEntry, Entry, ())
```

```
    (HeightEntry, SvalueResponder, (S_clear))
    (HeightSel, Container,())
        (CentimeteressItem, SvalSelector,())
        (InchesItem, SvalSelector,())
    (CaculateButton, ActionControl, (S_result, I_Calculate))
    (ClearButton, ActionControl, (S_Cleare))
    (QuitButton, ActionControl, ((QuitApp))
BMI_Result _Win is
    (Result, SvalueResponder, (S_result)
    (CloseButton, ActionControl, (I_Close))
```

Once we can determine the interaction behaviors from the code, we can easily build the full PIMs (see figure 7.10)

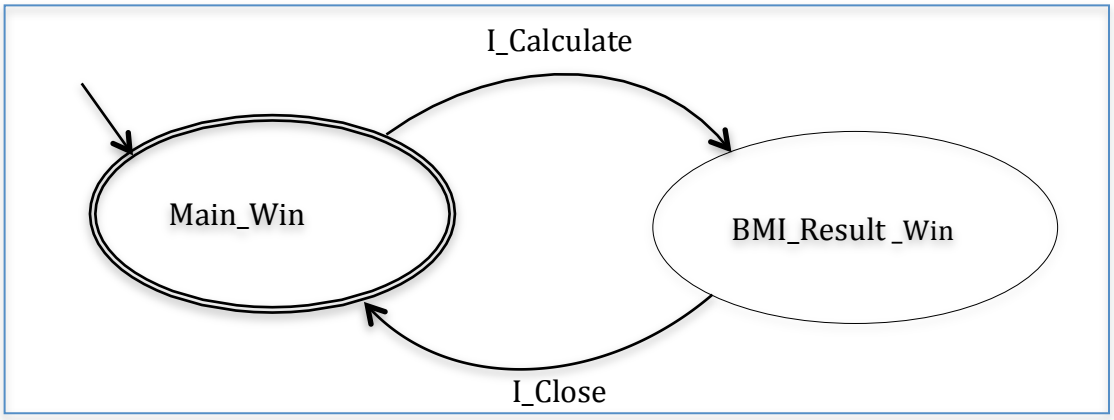


Figure 7.10. PIM for "BMI Calculator" app.

Thus, identifying the meaning of the statement in code is an important step to detect the information about the widgets in terms of a label and action of a widget and also helps to detect widget embedding to discover the relationship between the GUI elements. Once all the desired information is gathered, the PModels and PIMs can be built. The goal of this section is to show how we can extract all the information required from the parser tree for building our models. For this purpose, the "BMI Calculator" example was used to further simplify the understanding. Next, some of the problems in the "Digital Parrot" application defined in the previous chapter will be resolved by tracking and traversing the parser tree.

Chapter VIII

Overview of Issues in an Advanced Example

The earlier chapter, showed how tracking and traversing the parser tree by using the walker method can help to extract the desired information needed to create PModels and PIMs. The main idea of our approach is to parse an interactive system into a parser tree and then walk that tree twice. The first tree walk defines all the identifier symbols in the program, and the second determines these symbols and computes expression types to extract all the desired information that can help to produce our desired models.

Using this technique can also help to understand the meaning of the statement in any interactive system whatever the code style. We use for this purpose the "Digital Parrot" application, introduced previously in Chapter 4. This program is written using the abstract factory pattern (see section 5.4 for more details of design patterns in general and the abstract factory pattern in particular). Figure 8.1 shows part of class diagram of the Digital Parrot program.

The correlations between the general Abstract Factory structure and the Digital Parrot example are:

- AbstractFactory => ParrotModelFactory
- ConcreteFactory => UserInterfaceManager
- AbstractProduct => NavigatorComponent and MainViewComponent
- ConcreteProduct=> TimelineNavigator, MapNavigator, TrailNavigator, TextFilterComponent, GraphViewComponent, and TableViewComponent
- Client => ParrotApp

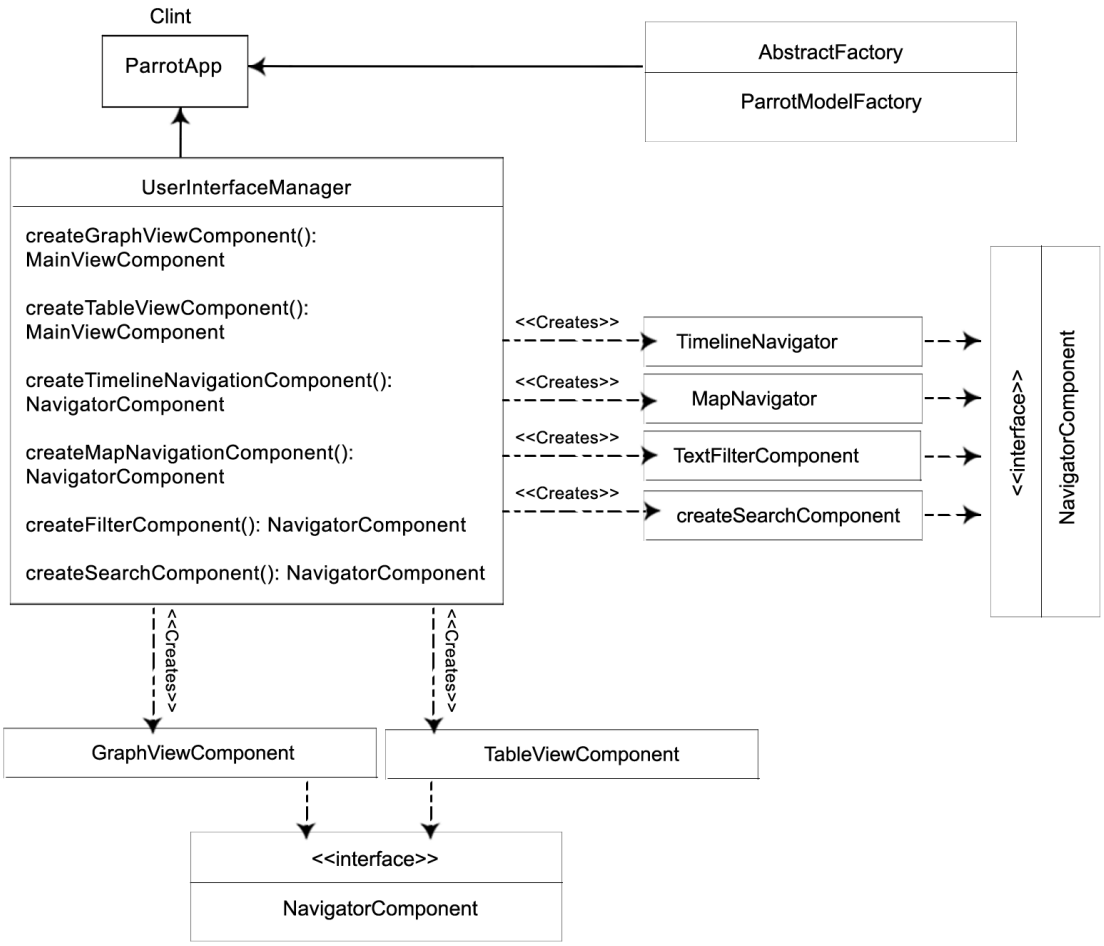


Figure 8.1. Part of class diagram of the "Digital Parrot" app

The "Digital Parrot" application is based on the inheritances between the classes in the application. This means a subclass inherits all the members (fields, methods, and nested classes) from its superclass/ parent class. As mentioned previously, the nested scope for classes was used to deal with class inheritance. Therefore, the symbol table covers all the information about the symbols and their scope correctly. However, due to the complexity of the "Digital Parrot" application in terms of code style and size, it is difficult to cover all the analysis process details to produce the PModels and PIMs. Thus, for clarity of understanding, an overview will be given of how our analysis process can work effectively with inheritance classes and show how we can solve some of the problems defined in section 5.4.

One of these problems is the difficulty to understand the composite value of constructor parameters, for example as in the following code segment, which is a part of "ParrotApp" class in the "Digital Parrot" program.

```
public class ParrotApp extends .JFrame
{
.
.
 private static final String APP_TITLE = "The Digital Parrot";
final boolean showTimeline = !line.hasOption(NO_TIMELINE_OPTION);
private UserInterfaceManager uiManager;
.
.
if (showTimeline) {
    NavigatorComponent timelineNavigator =
uiManager.createTimelineNavigationComponent();
    navigators.add(timelineNavigator);

JFrame timelineFrame=new JFrame(timelineNavigator.getNavigatorName()
+ ",Äì" + APP_TITLE);

    .
    .
   }
.
.
}
```

The main idea of the statement above is for creating a convenient title for the frame/window "***timelineFrame***", when clicking on the "***Time line***" JToggleButton. The code excutes this by using an "*if statement*", where "***showTimeline***" boolean gets the value based on the chosen frame/window option of GUI command line; where launching the "Digital Parrot" program relies on the interactive communication with a user at a command line. This only shows (not explains the implementation of this code) how to deal with the inheritance between classes by using the parser tree and extracting the composite value of the constructor parameter to get the correct value of the variable "***timelineFrame***", where the result value will be the title of the frame in the GUI.

To understand how to extract this information, all the formulae responsible for giving the desired title of the window in this code must be known. The fragments of these related code snippets are shown below.

```
   public class TimelineNavigator extends AbstractNavigatorPanel {
...
private static final String NAME = "Timeline";

  public String getNavigatorName() {
  return NAME;}
 ...
}
```

```
public interface NavigatorComponent {
 public String getNavigatorName();
...
 }
```

Based on this information, Figure 8.2 shows the class diagram and how the variables interact between the class and their methods.

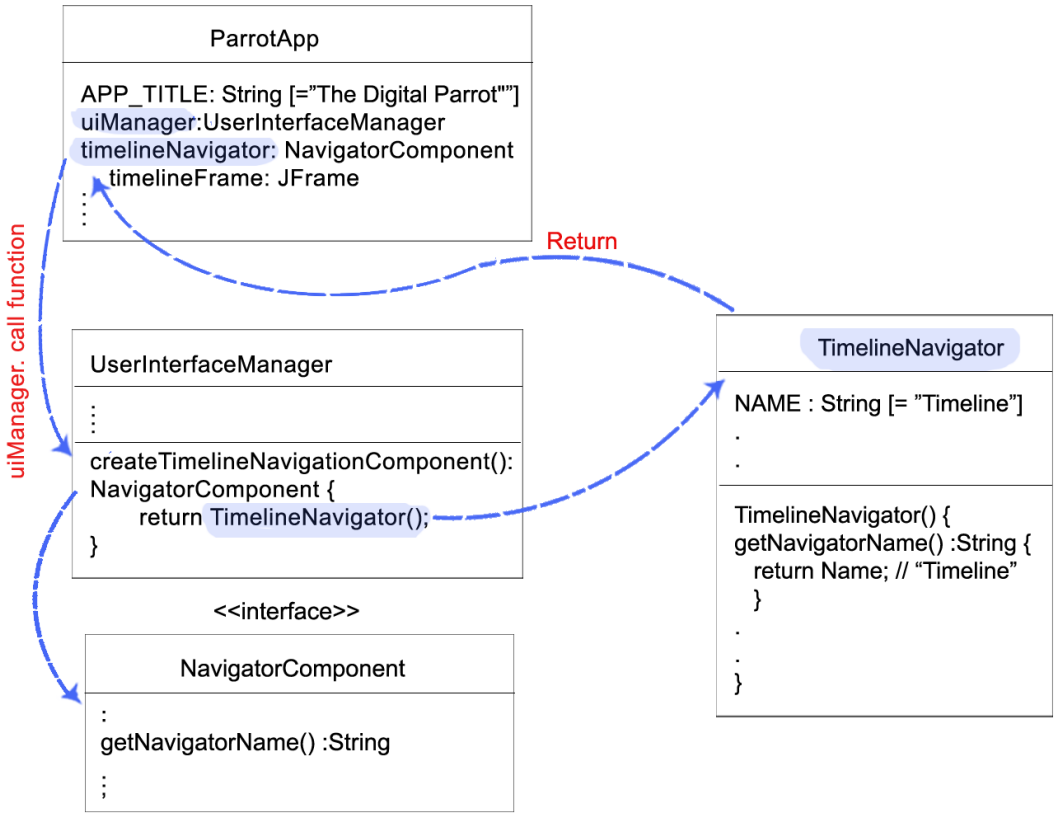


Figure 8.2. Interaction between the variables and methods in "Digital Parrot" classes

The dashed lines represent the interaction of the variables to call methods and return the values. The variable "***uiManager***" in ***ParrotApp*** class calls the method "***createTimelineNavigationComponen***t()" to return "***TimelineNavigator***()" as a value in the variable "***timelineNavigator***". Thus, "***timelineNavigator***" can use all the methods and variables in ***TimelineNavigator*** class.

The symbol table is able to extract all the symbols, variables, functions, type declarations, and initial values, associated with their scope. Figure 8.3 presents the output of defining symbols in the symbol table of the fragments above.

```
Define class :ParrotApp its superclass: javax.swing.JFrame:

     Defn field:APP_TITLE, of type: String, Value:[”The Digital Parrot”]
     ...
     Defn field:line, of type: CommandLine
     Defn field:datafile, of type: String
     Defn field:showTimeline, of type: tBOOLEAN
     ...
     locals:[line, datafile, showTimeline, ...]
     Defn field:timelineNavigator, of type: NavigatorComponent
     Defn field:timelineFrame, of type: JFrame
     locals:[timelineNavigator, timelineFrame]
     ..
Class ParrotApp:{APP_TITLE,...}
Define class :TimelineNavigator:
     ...
     Defn field:NAME, of type: String, Value:["Timeline"]
     Define Function :getNavigatorName
     Function <TimelineNavigator.getNavigatorName:tSTRING>
     ...
Class TimelineNavigator:{NAME, getNavigatorName,..}
     Define interface:NavigatorComponent:
     Function <NavigatorComponent.getNavigatorName:tSTRING>
     ...
interface NavigatorComponent:{getNavigatorName,..}
Define class :UserInterfaceManager:
     Constructor UserInterfaceManager.UserInterfaceManager:
     Function <UserInterfaceManager.createTimelineNavigationComponent:NavigatorComponent>
     ..
Class UserInterfaceManager:{UserInterfaceManager, createTimelineNavigationComponent}
...
```

Figure 8.3. A textual excerpt of a symbol table for the "Digital Parrot" app.

Once all the symbols in the tables are defined, they can easily be resolved by using the resolve () method. Figure 8.4 illustrates the output of resolving part of symbols in the symbol table.

```
[@370,1867:1875='APP_TITLE',<100>,52:29]cymbol_name: APP_TITLE,
<ParrotApp.APP_TITLE:String:[”The Digital Parrot”]>
[@406,2084:2092='uiManager',<100>,59:30]cymbol_name: uiManager,
<ParrotApp.uiManager:UserInterfaceManager>
...
[@623,3552:3555='line',<100>,103:15]cymbol_name: line,
<locals.line:CommandLine>
[@653,3774:3781='datafile',<100>,111:10]cymbol_name: datafile,
<locals.datafile:String>
[@690,3942:3953='showTimeline',<100>,116:17]cymbol_name: showTimeline,
<locals.showTimeline:tBOOLEAN>

[@1288,7189:7205='timelineNavigator',<100>,220:23]cymbol_name: timelineNavigator,
<locals.timelineNavigator:NavigatorComponent>

[@1304,7311:7323='timelineFrame',<100>,223:11]cymbol_name: timelineFrame,
 <locals.timelineFrame:JFrame>

[@2284,23462:23465='NAME',<100>,730:35]cymbol_name: NAME,
<TimelineNavigator.NAME:String:["Timeline"]>

[@2284,3011:3055='createTimelineNavigationComponent',<100>,500:247]cymbol_name:
 createTimelineNavigationComponent-> it is a Function
 <UserInterfaceManager.createTimelineNavigationComponent:tSTRING>:[]
...
```

Figure 8.4. Part of resolving output of the symbol table for the “Digital Parrot” app.

After extracting all the symbols in the program, we can track and traverse the parser tree to understand the meaning of the code’s statements to extract the required information. Using the code fragment shown below, the title of the frame from the widget constructor is to be extracted.

```
timelineFrame =new JFrame(timelineNavigator.getNavigatorName()+",Äì"
+ APP_TITLE);
```

Here, the variable’s value represents the content of the parameter, which has a composite value that needs to resolve the method call and then calculate all the text values of this parameter to convert the result to the variable's value.

To achieve this, the rule that matches a constructor in the parser tree needs to be determined (as in the previous chapter). The output parser tree of segment above is shown in Figure 8.5.

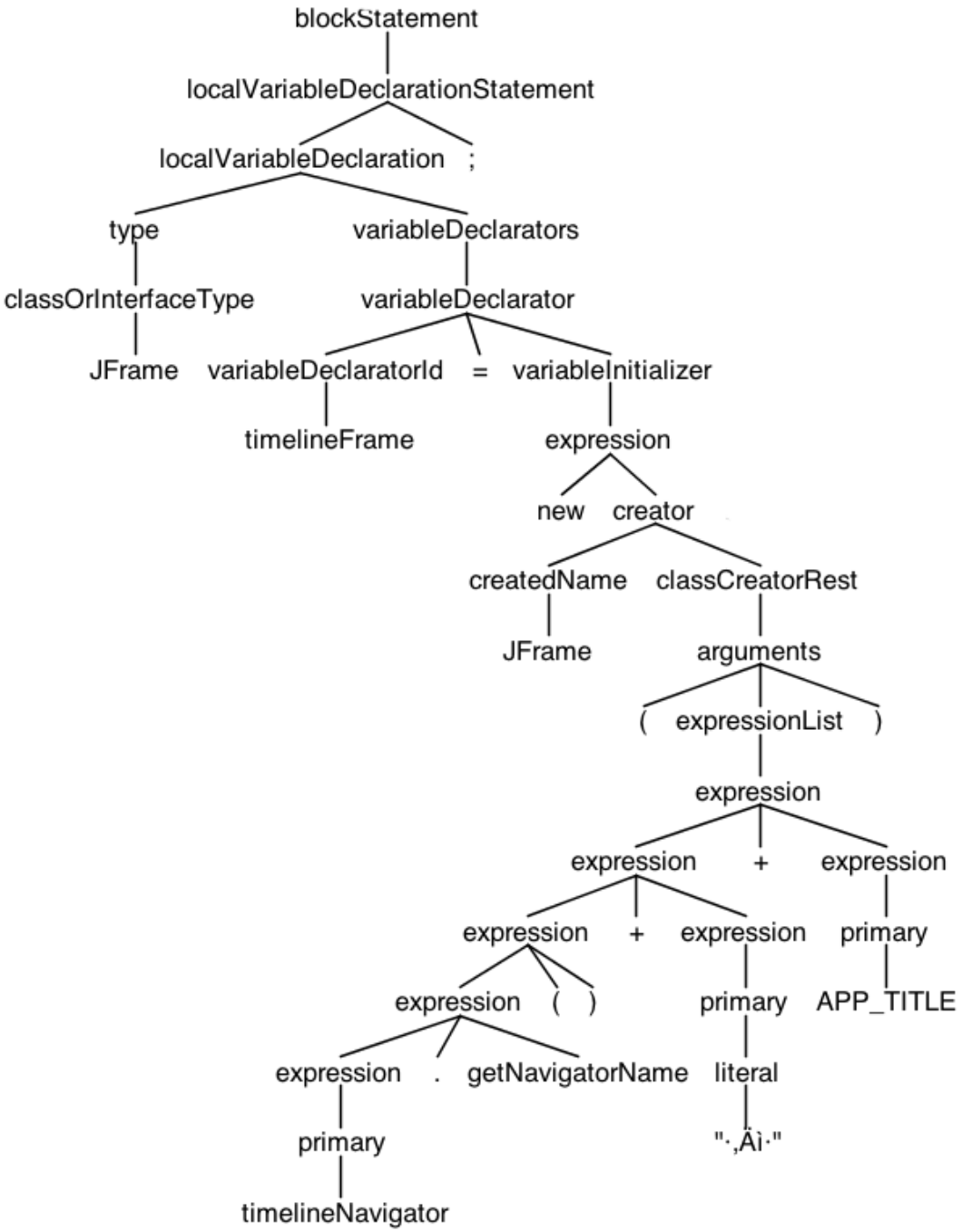


Figure 8.5. Output of parser tree for segmentation code.

It is now possible to extract the value of the symbol "***timelineFrame***" by implementing listener methods for the alternative rules. In the example parser tree above are many nodes but the focus is on those, which matter to get the value from

the addition process. So a listener method is needed for nodes that are responsible for dot phrases to obtain the value of the method and addition phrase to obtain the value of the two sub-expression children and then transfer the result to the symbol and store them in memory. The new code is shown below.

```
// -------- For id= varianleinitializer ---------
  public void exitExpreF(JavaParser.ExpreFContext ctx) {
    String valLeft=ctx.getChild(0).getText(); // ID
    Symbol symbol = currentScope.resolve(valLeft);
     ...
    if (ctx.getChild(1).getText().equals("="))
      if(!(stack.empty()))
    // Store the result in our memory value
     memory_value.put(symbol, stack.pop());
    }
// -------- push the operands in a stack ---------
public void exitPrimary(JavaParser.PrimaryContext ctx) {
 if (ctx.Identifier() != null)
 {
   Symbol sym = currentScope.resolve(ctx.Identifier().getText());
   if (sym!=null )
    if (sym instanceof VariableSymbol )
       System.out.println("There is not such a  symbol:"+
ctx.Identifier().getText()));
      else
      stack.push(sym .value); // push the value of a variable
    }
    if (ctx.literal() != null)
       stack.push(ctx.literal().getText());// push a number or String
}

// -------- For expression ('+'|'-') expression  ---------
  public void exitAddSubexpre(JavaParser.AddSubexpreContext ctx){

    String op1 = stack.pop();
    String op2 = stack.pop();
    String R=;

    if (ctx.getChild(1).getText().equals("+"))
    {
      R=op2 + op1;
      stack.push(R);
      System.out.println("-("+op1+") + ("+op2+")="+ R);
      System.out.println("Push The result to stack="+R);
    }
 }
```

Basically, to resolve symbol " ***timelineFrame*** " it is necessary to find the name in the current scope's symbol table, and store the variable with the final result of the calculation process. The primary () list is used to obtain only the elements of type

PrimaryContext. By this, we can determine whether they are an identifier or a string literal to store their values in a stack to pop them in addition phrases to calculate them. The output of this is shown in Figure 8.6 below.

```
-( - )+ ( Timeline)=Timeline -
Push The result to stack=Timeline -

-(APP_TITLE) + (Timeline -)=Timeline - The Digital Parrot
Push The result to stack= Timeline - The Digital Parrot

From the memeory, we have the final result, which is:
   timelineFrame = Timeline - The Digital Parrot
```

Figure 8.6. Output of part of segmentation code.

As shown in above Figure, we are able to solve the problem of extracting the composite value and transferring the result to the symbol/variable. At the moment, we are trying to investigate and analyze other parts in the "Digital Parrot" system to extract the required GUI models. This will be discussed in future work (section 9.4).

Chapter IX

Summary and Conclusion

This chapter summarizes the goal of this thesis and describes how it has been attained. It ends by giving some direction ideas of future work.

9.1 Overview of study goals

This project has presented a reverse engineering approach that abstracts information from the legacy source code of interactive systems. Most of the existing reverse engineering techniques give high priority to describing GUI aspects that mostly cover understanding the structure and execution behaviors of the interactive system, and ignore its corresponding internal behaviors. Unfortunately this is not conducive to a full analysis of the system; it causes difficulties in proving properties about the whole system to ensure that the software does the correct thing in all conditions. This research has discussed our approach of using reverse engineering techniques for interactive systems to extract both structural and functional aspects of the underlying system in order to create formal models: *Presentation Models* (of structure and functionality behaviors) and *Presentation Interaction Models* (of interactive behavior) [Bowen06].

9.2 Summary

This work has described a number of experiments and investigations, which aimed to discover and address all the possible problems and improve the analysis process in this area. For example, our initial experiments started with investigating whether clone detection can assist in our analysis process to reduce the complexity of PDGs. However, this experiment has shown that clone detection is not helpful in our area.

Then, trying to analysis the PDGs to extract the required models was one of the approaches used during this study, but there was missing information standing as an obstacle to build the models. Thus, this work then progressed to trying to combine the final model results from both static and dynamic methods to solve this problem. This attempt also, unfortunately, was unsuccessful in achieving the full models due to the lack of sufficient information to connect between the results of models in both approaches. During this journey, we suggested analysing the source code to detect all GUI elements and their behaviors to create PModels and PIMs. While this experiment was fairly good and solved the previous experiments' problems, it could not build the models of complex and advanced interactive application examples using one of the common object-oriented design patterns.

During all these experiments, we have focused on the practical side to extract the models rather than just suggesting theory and analysis to find solutions. Explorations in the search for solutions in most of these experiments contributed to finding our ultimate approach to this study.

9.3 The final approach

Our approach is based on parsing an interactive system source code into a parser tree and then walking that tree twice. The first tree walk defines all the identifier symbols in the program and stores them into a symbol table, and the second resolves these symbols and computes expression types to understand the meaning of each phrase in code to extract all the desired information that required to build our models. ANTLR tool is used for this purpose, where it produces the parser tree based on Java grammar and the tree-walking mechanisms in an automatic manner.

This work has examined our case study – "BMI Calculator" application - throughout this project and shown how the full models can be generated successfully from its parser trees, it also has tried to solve some issues associated with style code. The "Digital Parrot" application (considered as a complex/advanced example) was chosen for this purpose, however we need further investigations in this area, this will be discussed in future work.

The approach has demonstrated its ability to derive both PModels, and PIMs from the parser tree. In the first step the models capture a user-oriented view of the interface with its internal functionality of the system. In the second step the models capture all the interaction aspects of the UI in an illustrative chart form. This enables us to ensure both the correctness of the design, and give adequate evidence of the quality of that design prior to, during and after implementation.

9.4 Future work

The next step for this research is to further investigate the extraction of the GUI models of the "Digital Parrot" system. It has been seen that the symbols are extracted from the "Digital Parrot" application and sorted into symbol tables. Giving the meaning of each statement by resolving the symbol and following the structure/syntax of a sentence in code is very helpful to extract the information required from the source code to build the PModels and PIMs. Our objective of this thesis has been to investigate the possibility of the approach. However, we still need more investigations in the "Digital Parrot" system and other advanced applications (particularly those that use design patterns) to understand the meaning of code statements in interactive software systems to derive the GUI models.

For this research we have built a prototype/proof of concept tool based on our approach described in chapters 6 and 7 to automatically create the PModels and PIMs; this involves several analyzers and tools constructed using the ANTLR parser-generator. Further developing this tool to support further automation of the methods described in this thesis would be a logical progression for this work.

In the future, conducting more case studies and extending the analysis to handle more complex user interfaces is important. This study has focused only a subset of Java Swing widgets. Thus, there is more work required to examine approaches for other programming languages and toolkits that support Graphic User Interfaces, in order to make the approach more generic. In addition, we should consider conducting further investigations in this area to assist our analysis process. For example, combining our approach with program dependence graphs for an interactive system may prove beneficial.

9.5 Final conclusion

Our proof of concept tool can be use to traverse the parser tree created by ANTLR using tree walker methods to extract the program entities. Our tool then tracks these entities to capture their expression and the relation between them to collect all the desired information to build both the PModel (the presentation model) and PIM (the presentation interaction model) of an interactive system. We have achieved the objective of analyzing reverse engineering techniques for interactive software applications and conducted a number of experiments in order to identify the specific problems that occur, and have proposed solutions to these problems.

References

[Alexander77] Alexander, C., Ishikawa, S., & Silverstein, L. A. (1977). *A pattern language*. NY, New York: Oxford University Press.

[Gehringer01] Gehringer, E.F. (2001). *Lecture 16: Events and inner classes* [PowerPoint slides]. Raleigh, North Carolina: University of NC State. Retrieved from http://people.engr.ncsu.edu/efg/517/f02/common/syllabus/lectures.html

[Baker95] Baker, B. S. (1995). On finding duplication and near-duplication in large software systems. *In Proceedings of the 2nd IEEE Working Conference on Reverse Engineering,* (*WCRE'95*) (pp. 86–95). Los Alamitos, CA: IEEE Computer Society.

[Baxter98] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. *In Proceedings of the 14th International Conference on Software Maintenance*, (*ICSM*) (pp. 368-377). Los Alamitos, CA: IEEE Computer Society.

[Belmabrouk12] Belmabrouk, A., & Messabih, B. (2012). The reverse engineering in oriented aspect "detection of semantics clones". *International Journal of Scientific & Engineering Research, 3*(5), 2229-5518. Retrieved from http://www.ijser.org

[Blackburn04] Blackburn, M., & Nauman, A. (2004). Strategies for web and GUI testing. *SPC-2004014-MC, version 1.0.* Herndon, Virginia: Software Productivity Consortium. Retrieved from http://www.knowledgebytes.net

[Bowen06] Bowen, J., & Reeves, S. (2006). Formal refinement of informal GUI design artefacts. *In Proceedings of the Australian Software Engineering Conference* ,(*ASWEC'06*) (pp. 221–230). Sydney, NSW, Australian.

[Bowen07] Bowen, J., & Reeves, S. (2007). Formal models for informal GUI designs. *Electronic Notes in Theoretical Computer Science, 183* (11), 57–72.

[Bowen08] Bowen, J., & Reeves, S. (2008). Formal models for user interface design artefacts. *Innovations in Systems and Software Engineering, 4*(2), 125-141.

[Bowen12] Bowen, J., & Reeves, S. (2012). Modelling user manuals of modal medical devices and learning from the experience. *In Proceedings of the 4th ACM SIGCHI symposium on engineering interactive computing systems,* (*EICS '12*) (pp.121-130). New York, NY: ACM. doi: 10.1145/2305484.2305505

[Chikofsky90] Chikofsky, E.J., & Cross, J.H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software, 5*(1), 13–17. doi:10.1109/52.43044

[Cortier07] Cortier, A., d'Ausbourg, B., & Aït-Ameur, Y. (2007). Formal validation of java/swing user interfaces with the event b method. *In Proceedings of*

the 12th international conference on Human-computer interaction: interaction design and usability, HCI'07 (pp. 1062-1071). Berlin: Springer-Verlag.

[Davey95] Davey, N., Barson, P., Field, S., & Frank, R. (1995). The development of a software clone detector. *International Journal of Applied Software Technology, 1*(3/4), 219-236.

[Doan08] Doan, T. (2008). *An evaluation of four reverse engineering tools for C++ applications*. (Master's thesis, University of Tampere, Tampere, Finland). Retrieved from http://tampub.uta.fi

[Evans07] Evans, W. S., Fraser, C. W., & Fei, M. (2007). Clone detection via structural abstraction. *Software Quality Journal, 17*(4), 309 – 330. doi: 10.1007/s11219-009-9074-y

[Foster98] Foster, H., Goradia, T., Ostrand, T., & Szermer, W. (1998). A visual test development environment for GUI systems. *ACM SIGSOFT Software Engineering Notes, 23*(2), 82 – 92. doi: 10.1145/271775.271793

[Freeman04] Freeman, E., Sierra, K., Freeman, E., & Bates, B. (2004). *Head first design patterns*. Sebastopol, CA: O'Reilly.

[Gimblett10] Gimblett, A., & Thimbleby, H. (2010). User Interface Model Discovery: Towards a Generic Approach. *In)of the 2nd ACM SIGCHI symposium on engineering interactive Ccomputing Systems*, (*EICS '10*) (pp. 145 – 154). doi: 10.1145/1822018.1822041.

[Gray98] Gray, J. (1998). *What Next? A Few Remaining Problems in Information Technology*.(Paper presented at the ACM Federated Research Computer Conference, Atlanta, GA). Retrieved from http://research.microsoft.com/pubs/68743/gray_turing_fcrc.pdf

[Hicinbothom93] Hicinbothom, J. H., & Zachary, W. W. (1993.). A tool for automatically generating transcripts of human-computer interaction. *Human Factors and Ergonomics Society Annual Meeting Proceedings, 37*(15), 1042.

[Higo07] Higo, Y., Kamiya, T., Kusumoto, S., & Inoue, K. (2007). Method and implementation for investigating code clones in a software system. *Information and Software Technology, 49*(9–10), 985–998.

[Huntington02] Huntington, D. (2002). Back to basics - backward chaining: an expert system fundamental. *PC AI Magazine,16*(4), 27-33. Retrieved from http://www.pcai.com

[Issa11] Issa, A., Sillito, J., & Garousi, V. (2011). Visual Testing of Graphical User interfaces: an Exploratpry Study Towards Systematic Definitions and Approaches. *In Proceedings of 14th IEEE International Symposium on Web Systems Evolution*, (*WSE*) (pp. 11 - 15). Los Alamitos, CA: IEEE Computer Society.

[Jacky97] Jacky, J. (1997). *The Way of Z: Practical programming with formal methods*. New York, NY: Cambridge University Press.

[Kuchana04] Kuchana, P. (2004). *Software Architecture Design Patterns in Java*. Boca Raton, FL: Auerbach.

[Kuhn01] Kuhn, D. R., Chandramouli, R., & Butler, R.W. (2001). *Cost effective use of formal methods in verification and validation* [CD-ROM]. Paper presented at the Workshop on Foundations for Modeling and Simulation (M&S) Verification and Validation (V&V) in the 21st Century, San Diego, CA.

[Lague97] Lague, B., Proulx, D., Merlo, E., Mayrand, J., & Hudepohl, J. (1997). Assessing the benefits of incorporating function clone detection in a development process. *In Proceedings of the IEEE International Conference on Software Maintenance* (pp. 314). Los Alamitos, CA: IEEE Computer Society.

[Lin12] Lin, F. (2012). *Analysing Reverse Engineering Techniques for Interactive Systems*, (Master's thesis, University of Waikato, Hamilton, New Zealand). Retrieved from http:// http://researchcommons.waikato.ac.nz/bitstream/handle/10289/6612/thesis.pdf

[Martinez11] Martinez, L., Pereira, C., & Favre, L. (2011). Reverse Engineering Activity Diagrams from Object Oriented Code: An MDA-Based Approach. *Computer Technology & Application, 2*(11), 969.

[Memon01] Memon, A., M. (2001). A comprehensive framework for testing graphical user interfaces. (Doctoral dissertation thesis, University of Pittsburgh). Retrieved from http://www.cs.umd.edu/~atif/papers/MemonPHD2001.pdf

[Memon03] Memon, A., Banerjee, I., & Nagarajan, A. (2003). GUI ripping: reverse engineering of graphical user interfaces for testing. In *proceedings of the 10th Working Conference on Reverse Engineering (*pp. 260-269). Los Alamitos, CA: IEEE Computer Society.

[Mesbah09] Mesbah, A., & Van Deursen, A. (2009). Invariant based automatic testing of AJAX user Interfaces. *In Proceedings o the IEEE 31st International Conference* on *Software Engineering, (ICSE 2009)* (pp. 210 – 220). Los Alamitos, CA: IEEE Computer Society.

[Mishne04] Mishne, G. & Rijke, M. (2004). Source code retrieval using conceptual similarity. *In Proceedings of the 2004 Conference on Computer Assisted Information Retrieval, (RIAO'04)* (pp. 539-554). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.7753&rep=rep1&type=pdf

[Morgado11] Morgado, I.C., Paiva,A.C.R., & Faria,J.P. (2011). Reverse engineering of graphical user interfaces. In *Proceedings of the ICSEA of the 6th*

International Conference on Software Engineering Advances (pp. 293-298). Los Alamitos, CA: IEEE Computer Society.

[Muller09] Muller, H., A., &Kienle, H., M. (2009). A small primer on software reverse engineering. *Technical Report, University of Victoria, Canada*. Retrieved from http://holgerkienle.wikispaces.com/file/view/MK-UVic-09.pdf

[Nau82] Nau, D. S. (1982). An Investigation of the Causes of Pathology in Games. *Artificial Intelligence 19(*3): 257-278.

[O'Brien05] O'Brien, L. (2005). *Reverse Engineering* [Powerpoint slides]., Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. Retrieved from http://www.cs.cmu.edu http://www.cs.cmu.edu/~aldrich/courses/654-sp05/handouts/MSE-RevEng-05.pdf

[Pacione03] Pacione, M. J., Roper, M. & Wood, M. (2003). A comparative evaluation of dynamic visualization tools. *In Proceedings of the 10th Working Conference on Reverse Engineering, (WCRE '03)* (pp. 80 – 89). Los Alamitos, CA: IEEE Computer Society.

[Paiva08] Paiva, A. C. R., Faria, J. C. P., & Mendes, P. M. C. (2008). Reverse engineered formal models for GUI testing. *In Proceedings of the Formal Methods for Industrial Critical Systems, 12th International Workshop, (FMICS)* (pp. 218-233). Berlin, Germany: Springer-Verlag.

[Paiva10] Grilo, A.M.P., Paiva, A. C. R., & Faria, J.R. (2010). Reverse Engineering of GUI Models for Testing. *In Proceedings of the 5th Iberian Conference on Information Systems and Technologies, (CISTI)* (pp. 1-6). Los Alamitos, CA: IEEE Computer Society.

[Parr09] Parr, T. (2009). *Language implementation patterns*.. Raleigh, NC: The Pragmatic Programmers. Retrieved from http://pragprog.com/

[Parr12] Parr, T. (2012). *The definitive antlr 4 reference*. Raleigh, NC: The Pragmatic Programmers. Retrieved from http://pragprog.com/

[Perry95] Perry, W. (1995). *Effective methods for software testing*. New York, NY: John Wiley & Sons. Retrieved from http://www.gbv.de/dms/ilmenau/toc/505007495.PDF

[Reeve05] Reeve, G. (2005). *A refinement theory for µCharts* (Doctoral dissertation University of Waikato, Hamilton, New Zealand). Retrieved from www.cs.waikato.ac.nz/pubs/2005/pdfs/reeve-thesis.pdf

[Rieger98] Rieger, M., & Ducasse, S. (1998). Visual detection of duplicated code. In *Proceedings of the Workshop ion on Object-Oriented Technology, (ECOOP '98),* (pp. 75-76). London, United Kingdom: Springer-Verlag.

[Roy07] Roy, C., K., &. Cordy, J., R. (2007). A survey on software clone detection research. *Queen's Technical Report, 541*, 115.

[Roy09] Roy, C., Cordy, J., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming, 74*(7), ~~pp.~~ 470–495.

[Schweer11] Schweer,A. (2011). *Augmenting Autobiographical Memory: An Approach Based on Cognitive Psychology Augmenting Autobiographical Memory: An Approach Based on Cognitive Psychology*. (Doctoral dissertation, thesis's, the University of Waikato, Hamilton, New Zealand). Retrieved from http://researchcommons.waikato.ac.nz

[Silva06] Silva, J. C., Campos, J. C., & Saraiva, J. (2006). Combining formal methods and functional strategies regarding the reverse engineering of interactive applications. *In Proceedings of the Interactive Systems: Design, Specification, and Verification. 13th International Workshop, (DSVIS)* (pp. 137-150). Berlin, Germany: Springer-Verlag.

[Silva10] Silva, J. C., Silva, C., Goncalo, R., Saraiva, J., & Campos, J. C. (2010). The GUISurfer tool: Towards a language independent approach to reverse engineering GUI code. *In Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, (EICS'10)* (pp. 181-186). New York, NY: ACM.

[Silva11] Silva, J. C., Saraiva, J., & Campos, J. C. (2011). Models for the reverse engineering of Jave/Swing application. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering, (ateM 2006).* Retrieved from http://news.informatik.uni mainz.de/ALT/Dateien/26-Silva-old.pdf

[Sommerville04] Sommerville, I. (2004). *Software Engineering*, (7th ed.). Harlow, United Kingdom: Pearson Addison Wesley.

[Staiger07] Staiger, S. (2007). Reverse Engineering of Graphical User Interfaces Using Static Analyses. In P*roceedings of the 14th Working Conference* on *Reverse Engineering,(WCRE)* (pp.189 – 198). Los Alamitos, CA: IEEE Computer Society.

[Systä00] Systä, T. (2000). *Static and dynamic reverse engineering techniques for java software systems*. (Doctoral dissertation, University of Tampere, Kalevantie, Finland). Retrieved from http://tampub.uta.fi/bitstream/handle/10024/67001/951-44-4811-1.pdf?sequence=1

[Systä99] Systä, T. (1999). On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software. . *In proceedings of the 6th Working Conference on Reverse Engineering* (pp. ~~P.~~304 – 313). Los Alamitos, CA: IEEE Computer Society. Atlanta, GA.

[Thimbleby06] Thimbleby, H. (2007). Interaction Walkthrough: Evaluation of safety critical interactive systems. *Interactive Systems. Design, Specification, and Verification: Lecture Notes in Computer Science, 4323*, 52 – 66.

[Walworth97] Walworth, A. (1997). Java GUI testing. *Dr. Dobb's Journal, 22(*2), 30.

Appendix A

The Source Code of the “BMI Calculator” in Java is shown here.

```
/**
 *
 * @author Alsharif Aman
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.TitledBorder;

public class MBI_calculater extends JFrame {
    private JLabel weightLabel;
    private JLabel weightLabel1;
    private JLabel HeightLabel;
    private JLabel HeightLabel1;
    private JPanel jPanel1;
    private JPanel jPanel2;
    private JPanel jPanel3;
    private JTextField weightTF;
    private JTextField HeightTF;
    private JFrame frame2 = new JFrame("Result");
    String str2, str;
    double res;
    int flage=0;
    String ImagIc[]= {"Images/a0.png", "Images/a1.png",
        "Images/a2.png", "Images/a3.png", "Images/a4.png"};
    String text[] = {" ","underweight", "normal weight", "
        overweight", "obese"};
    ImageIcon imageIcon = new
         ImageIcon(getClass().getResource(ImagIc[0]));

   JLabel ResLabel = new JLabel( imageIcon);
   JLabel Titel= new JLabel();
   // End of variables declaration
   //----------------------------------
    public  MBI_calculater()
    {
        super("MBI_calculater");
        jPanel1 = new JPanel();
        jPanel1.setLayout(null);
        jPanel1.setBounds(new Rectangle(8, 14, 485, 200));
        TitledBorder titled = new TitledBorder("Enter your
        personal infromation:");
        jPanel1.setBorder(titled);

        // The result will be in a new Frame :

        frame2.setSize(500, 350);
        frame2.setTitle("BMI Result");
        frame2.setLocation(200, 100);
        frame2.setLocationRelativeTo(null);
        jPanel2 = new JPanel();
```

```
jPanel3 = new JPanel();
weightLabel = new JLabel("Current weight is :");
weightLabel.setBounds(15,40,120,20);
jPanel1.add(weightLabel);
weightTF = new JTextField();
weightTF.setBounds(140,40,60,20);
jPanel1.add(weightTF);
weightLabel1 = new JLabel(); // show example
weightLabel1.setBounds(140,60,120,20);
weightLabel1.setFont(new Font("Arial",Font.BOLD, 10));
weightLabel1.setForeground(Color.BLUE);
weightLabel1.setText("e.g: 58 kg  or 128 pound");
jPanel1.add(weightLabel1); // to Appear

final String W[] = {"kilograms","pounds"};
final JComboBox WCB = new JComboBox(W);
WCB.setBounds(220,40,125,20);
jPanel1.add(WCB);
WCB.addItemListener(new ItemListener(){
public void itemStateChanged(ItemEvent ie){
str = (String)WCB.getSelectedItem();

      }
 });

 HeightLabel = new JLabel("Current Height is :");
 HeightLabel.setBounds(15,90,120,20);
 jPanel1.add(HeightLabel); // to Appear

 HeightTF = new JTextField();
 HeightTF.setBounds(140,90,60,20);
 jPanel1.add(HeightTF); // to Appear in the pabel

 HeightLabel1 = new JLabel();
 HeightLabel1.setBounds(140,110,170,20);
 HeightLabel1.setFont(new Font("Arial",Font.BOLD,10));
 HeightLabel1.setForeground(Color.BLUE);
 HeightLabel1.setText("e.g: 64 inches  or 160
 centimeters");
 jPanel1.add(HeightLabel1);
 final String H[] = {"inches", "centimeters"};
 final JComboBox HCB = new JComboBox(H);
 HCB.setBounds(220,90,125,20);
 jPanel1.add(HCB); // to Appear
 HCB.addItemListener(new ItemListener(){
 public void itemStateChanged(ItemEvent iee){
 str2 = (String)HCB.getSelectedItem();

          }
            });

JButton ClcBtn = new JButton("Calculate");
ClcBtn.setBounds(50,160,110,20);

ClcBtn.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e)
    {
```

```
 frame2.setVisible(true);
 final double t1 ,t2;
 double temp1, temp2;
 ResLabel.setText(" ");
 Titel.setText(" ");
 ResLabel.setIcon( null ); // to cleare the icon

 if(  str2 == H[0])  //=  inch
   {
    temp2 =
  (double)(Double.parseDouble(HeightTF.getText())*
  2.54);   // convert inches to centimeters
  t2 = (temp2 * 0.01); // convert cm to m .

  } else {
     t2 = Double.parseDouble(HeightTF.getText()) * 0.01;

         }
if(  str == W[1]) //  =pound
   {
   t1 = (Double.parseDouble(weightTF.getText()) /
  2.2);// convert pound to Kg
  }else {
     t1 =  Double.parseDouble(weightTF.getText());
        }
  temp1=t2*t2;
  res= t1 / temp1;

  Titel = new JLabel(" >> your Body Mass Index : " + res
  + " . ");
  Titel.setBounds(8, 10, 485, 200);

  jPanel2.add(Titel); // to Appear
   Titel.repaint();

  if( res < 18.5)
    {
    flage=1;

    }else if( (res >18.5) && (res < 24.9))
     {

     flage=2;

  } else if( (res >25) && (res < 29.9))
      {
        flage=3;

       } else if(res > 30)

        { flage=4;

        }
  if (flage !=  0)
  {
     imageIcon = new
  ImageIcon(getClass().getResource(ImagIc[flage]));
     ResLabel = new JLabel( imageIcon);
```

```
            ResLabel.setBounds(10, 100, 450, 200);
            ResLabel.setFont(new Font("Arial",Font.BOLD, 20));
            ResLabel.setForeground(Color.RED);
            ResLabel.setText("The " + text[flage] + "
        range..");
            ResLabel.setHorizontalTextPosition(JLabel.LEFT);
            ResLabel.setVerticalTextPosition(JLabel.CENTER);
            jPanel2.add(ResLabel);
            ResLabel.repaint();

        }

            }
        });
          jPanel1.add(ClcBtn);
          JButton closeButton = new JButton("Close");
          closeButton.setBounds(190, 280, 110, 20);
          closeButton.addActionListener(new ActionListener() {
              public void actionPerformed(ActionEvent event) {
                Close_fun();
              }
            });
            jPanel3.add(closeButton);

           JButton ClButton = new JButton("Clear");
           ClButton.setBounds(190,160,110,20);
           getRootPane().setDefaultButton(ClButton);
           ClButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
               weightTF.setText(" ");
               HeightTF.setText(" ");
              }
            });
            jPanel1.add(ClButton); // to Appear in the pabel

            JButton quitButton = new JButton("Quit");
            quitButton.setBounds(330, 160, 110, 20);
           quitButton.addActionListener(new ActionListener() {
              public void actionPerformed(ActionEvent event) {
                Quit_App();
              }
            });
            jPanel1.add(quitButton);

            add(jPanel1);
            frame2.add(jPanel2);
            frame2.add(jPanel3);
            }
           public void Close_fun()
            {
              frame2.setVisible(false);
            }
           public void Quit_App()
            {
               System.exit(0);
             }

public static void main(String[] args)
```

```
    {
        MBI_calculater BMI_mainWin = new MBI_calculater();
        BMI_mainWin.setSize(500, 250);
        BMI_mainWin.setTitle("BMI Calculator");
        BMI_mainWin.setLocation(200, 100);
        BMI_mainWin.setLocationRelativeTo(null);

        BMI_mainWin.setDefaultCloseOperation(JFrame.EXIT_ON_CL
        OSE);

        BMI_mainWin.setVisible(true);
    }

}
```