Abstract

WYSIWYG (What You See Is What You Get) graphical editors, for example, Swing Designer and Dreamweaver, are widely used to help GUI developers in developing and maintaining GUIs (Graphical User Interface) of desktop and Web applications. GUI editors allow developers to work directly with a graphical design view instead of scattered source elements. This feature helps developers in addressing one particular difficulty for GUI development and maintenance: a large amount of GUI code can be scattered across many different source modules. However, traditional GUI editors have two major defects. First, GUI code is normally tangled with some dynamic computation code. Traditional GUI editors are limited by their ability to statically reconstruct this dynamic GUI code, creating an incomplete design view for developers. Second, some parts of a user interface are stateful and reactive. Their appearance and behavior vary over time, based on mutations of state made from code. Currently, there are no existing GUI editors that can assist developers in understanding how a UI behavior is implemented. To deal with the first defect, I built a tool called FreezeFrame. This tool uses a dynamic reverse-engineering approach to bridge the gap between the rendering view of a user interface and its corresponding implementation, by providing a design/code hyper-linking on a complete dynamic view. To deal with the second defect, I created a tool called ScriptInsight. This tool provides a custom control flow model to bridge a live UI behavior with a set of source elements that control the UI behavior. Developers can use this model to understand the implementation of a UI behavior. To evaluate FreezeFrame, I first show that user interface code can be spread across the decomposition of applications. Next, I demonstrate that the reverse-engineering capabilities of a traditional GUI editor can be improved by using my dynamic reverse-engineering approach. To evaluate ScriptInsight and the usefulness of the custom control flow model, I first evaluate this tool by presenting several case studies. Second, I justify the relevance of this model by the complexity of JavaScript implementations for the UIs from several existing Web applications.
Preface

Chapter 3 of this thesis is based on the following paper,

Chapter 4 of this thesis is based on the following paper,

Both of the two papers were co-authored with my supervisor Eric Wohlstadter. I am the first author of the papers. For the writing of this thesis, I have used the first person narrative to reflect my own ideas of the research. Eric's mentorship provided me with guidance in refining my ideas and copy editing.
# Table of Contents

Abstract .......................................................................................................................... ii
Preface ............................................................................................................................ iii
Table of Contents ............................................................................................................. iv
List of Tables .................................................................................................................. viii
List of Figures ................................................................................................................ x
Glossary .......................................................................................................................... xviii
Acknowledgements ......................................................................................................... xix
Dedication ....................................................................................................................... xx

Chapter 1 Introduction .................................................................................................. 1
  1.1 Problem Domain .................................................................................................. 2
    1.1.1 Existing GUI Editing Tools ........................................................................... 3
    1.1.2 Two Major Defects of Traditional GUI Editors ........................................... 4
  1.2 Research Objectives and Contributions ............................................................. 5
    1.2.1 Defect 1: Limited Reverse-Engineering Capabilities .................................... 5
    1.2.2 Defect 2: No Explicit Support for Coordinated Behaviour ......................... 9
    1.2.3 Contributions ............................................................................................... 13

Chapter 2 Background .................................................................................................. 14
  2.1 Structured GUIs, Widgets and GUI Design ......................................................... 14
  2.2 GUI Editors ......................................................................................................... 15
    2.2.1 Forward Engineering .................................................................................... 16
    2.2.2 Reverse-Engineering ................................................................................... 16
  2.3 GUI Events and Event Handlers .......................................................................... 17
  2.4 GUI Developers .................................................................................................. 17
List of Tables

Table 3.1: The subset of the Java language specification contains all Java terms that are important to my research. The Java terms are listed in the first column of this table. The second column describes the understanding of these terms from the GUI domain. ..............................................................27

Table 3.2 [Example]: A list of types and methods from the code example in Figure 3.8, which have been hard-coded into the GUI editor .................................................36

Table 3.3: Simplified pointcuts for FreezeFrame and corresponding explanation...............51

Table 3.4: Application subjects and their sizes in terms of non-commented lines of code....66

Table 3.5: This metric measures the number of classes containing GUI join points captured by GUIAspect for rendering of the main window view........................................67

Table 3.6: Size metric is determined by the number of widgets, and set widget properties used in the dynamic model and the static design model...............................68

Table 3.7: Size of design information displayed by original and augmented GUI editor (FreezeFrame) compared to total amount of information in the dynamic view....70

Table 4.1: The complete history trace (\(tc\)), the abstract trace (\(atc\)), and the DMG representation (\(DMG\)) of the accordion row behaviors..............................................86

Table 4.2: Relevant HTML elements for the JPS 2 examples. The corresponding properties are dynamically mutated to create various effects described in this section. The notation (2) shows that the property has been mutated twice, and the (*) shows that the property has been mutated in a loop.................................................................90

Table 4.3: (# of Files) lists the number of JavaScript files downloaded for each page and (Total Lines) is the sum of their file line counts......................................................108

Table 4.4: (Average Context) lists the average number of distinct contexts that mutator statements were executed in. (Standard Deviation) lists the corresponding standard
deviation. (Total Mutator) lists the total number of DOM mutator statements executed for the page.

Table 4.5: (Memory Without ScriptInsight) lists the memory usage of Firefox with a page loaded, after having its UI exercised, without my tool in use and (Memory With ScriptInsight) lists the results with the use of my tool.
List of Figures

Figure 1.1 [Example]: Screenshot from Java-Chess program execution. Sixteen classes contribute to this dynamic view..................................................................................................................6

Figure 1.2 [Example]: Screenshot from Swing Designer GUI editor for Java-Chess class. Several widgets are missing from the recovered design view. Extra rulers and grid markers are the effect of the editor tool in action.................................................................7

Figure 1.3 [Example]: Java-Chess source code that is used to create the column panel of the chess board. Code has been refactored to improve readability. ...........................................9

Figure 1.4 [Example]: A snapshot from the Catalog Browser page of Java Pet Store 2. The information pane is closed and only displays the name, rating, price and short description of the pet. The collapse button is an up arrow, to indicate that this button can be clicked to open this information pane..................................................10

Figure 1.5 [Example]: A snapshot from the Catalog Browser page of Java Pet Store 2. The information pane is fully opened and now displays the name, rating, price, short description, and full description of the pet. The collapse button is a down arrow, to indicate that this button can be clicked to close the information pane......................11

Figure 2.1: Forward- and reverse-engineering process for graphical user interface editors...16

Figure 3.1 [Example]: The dynamic view for the main window of the Java-Chess program. The intended editing tasks are marked by: a) the “3 seconds” menu item, b) a button panel, and c) the column labels of the chessboard.........................................................20

Figure 3.2 [Example]: Screenshot from Swing Designer GUI editor for Java-Chess class. Several widgets are missing from the recovered design view. Extra rulers and grid markers are the effect of the editor tool in action...............................................................22

Figure 3.3 [Example]: Adding a new “Hint” button in the developer’s intended position of the NavigationPanel..........................................................23
Figure 3.4 [Example]: The added new “Hint” button is squeezed to the right side of the navigation panel at runtime because the developer did not consider the context of the panel’s container. ................................................................. 24

Figure 3.5 [Example]: The ChessBoardRenderer2D class is opened by using the GUI editor. All the labels for the columns and rows cannot be rendered because of the limited reverse-engineering capabilities of the GUI editor. ........................................... 25

Figure 3.6: The algorithm used to identify the root method of a given visual class. ............ 30

Figure 3.7: Analyzing the identified root method and generating one set that contains all the identified widgets (widgtSet) and another set that contains all the identified widget hierarchy relationships (relationshipSet). ................................................. 33

Figure 3.8 [Example]: This code snippet is used to illustrate the code analyzing technique used by traditional GUI editors. This code creates two custom widgets through a custom widget class, TextJButton. Those two custom buttons are added onto a frame. ........................................................................................................... 35

Figure 3.9 [Example]: The design view (a) and widget hierarchy tree model (design model) (b) for the GUI code in Figure 3.8. The design view is rendered by using Swing Designer. In the design model, each node contains the object name and class name, which are separated by using a colon................................................................. 36

Figure 3.10 [Example]: The chess board generated by ChessBoardRenderer2D class. The columns (called files) are labeled by the letters a to h and the rows (called ranks) by the numbers 1 to 8................................................................. 38

Figure 3.11 [Example]: The constructor code of ChessBoardRenderer2D class. The highlighted code snippet (a) generates the row numbers labeled 1 to 8. The highlighted code snippet (b) generates the chess board. The highlighted code snippet (c) generates the column numbers labeled a to h......................................................... 39

Figure 3.12 [Example]: The implementation for a custom JFrame. Three buttons are arranged on the custom frame (CustomizedFrame) with a layout manager, GridLayout. The title string of the frame is passed into its constructor from another class, testCustomizedFrame................................................................. 41

Figure 3.13 [Example]: The rendered design view for the source code as shown in Figure 3.12 by a traditional GUI editor. Three buttons are arranged on the frame in a grid
layout. The missing frame title is replaced with the tag <dynamic> because the GUI editor lacks the title string and cannot display it. .................................................42

Figure 3.14 [Example]: The implementation for a custom frame. Three buttons are arranged on the custom frame by using the layout manager that is passed into the constructor of this custom frame class (CustomizedFrame). The frame title is fixed in line 4. .........................................................................................43

Figure 3.15 [Example]: The rendered design view for the source code as shown in Figure 3.14 by the same traditional GUI editor. Three buttons are missing from the frame because the GUI editor lacks the layout information. .............................................43

Figure 3.16 [Example]: The code example demonstrates the usage of function-based code abstraction. The two text buttons are created by using a helper method, createTextButton. .........................................................................................46

Figure 3.17 [Example]: The design view for the code in Figure 3.16. The frame is rendered, but the two buttons are missing because the traditional GUI editor has limited support for function-based code abstraction.........................................................47

Figure 3.18 [Example]: The widget hierarchy tree for the Java-Chess user interface. All the nodes represent the widgets in the Java-Chess user interface. The solid lines represent static relationships that can be recovered by traditional GUI editors. The dotted lines represent dynamic relationships that can only be recovered at runtime. .................................................................................................48

Figure 3.19 [Example]: The truncated widget hierarchy tree. Compared to the tree in Figure 3.18, all the widgets directly connected to the tree model through dynamic relationships and all the corresponding child nodes of the widgets (not reachable from the root widget) are deleted. This truncated widget hierarchy tree represents all the widgets that can be displayed by traditional GUI editors.........................49

Figure 3.20: During the execution of a GUI application, a developer chooses a specific user interface that they are interested in. Then the dynamic model of the user interface is captured by using the advice of a monitoring aspect. Next, the static design model that is normally used by a GUI editor is replaced with this dynamic model. ........................................................................................................50
Figure 3.21 [Example]: Java-Chess source code in the constructor of chess board column panel. Code has been refactored to improve readability.................................53

Figure 3.22 [Example]: A developer changed the label ‘g’ to the label ‘G’ in the design view using FreezeFrame, if it naively translated line 5 to $\text{setText("G")}$, we would end up with eight identical labels.................................................................53

Figure 3.23: This editing constraint used by my tool. This constraint is used to determine if a widget on a design view can be edited safely.......................................................56

Figure 3.24 [Example]: The static design model of $\text{ChessBoardRenderer2D}$ class, which has been recovered by Swing Designer. In each node of the tree, the name of the widget and the corresponding type information are presented and separated by a colon.................................................................................................57

Figure 3.25 [Example]: A sub-tree of the dynamic model for the main window of the Java-Chess application (rooted at the $\text{chess Board}$ widget). In each node, the name of the widget and the corresponding type information are presented and separated by a colon........................................................................................................58

Figure 3.26 [Example]: High-level illustration of a dynamic model (object graph) including design (solid) and dynamic (dotted line) information. Numeric labels refer to “add” relationships between widgets in Figure 3.27. A design model will appear to the user as a truncated version of a dynamic model......................................................59

Figure 3.27 [Example]: Three custom widget classes used to demonstrate how dynamic context improves view-editing (an illustration is shown in Figure 3.26). Notice that, since the argument in the $\text{MyFrame}$ constructor requires knowledge of the dynamic type, it cannot be displayed in a traditional design view.......................60

Figure 3.28 [Example]: A complete design view of the Java-Chess main window. All the missing widgets are recovered by using my analysis. Sub-trees lost in Figure 3.2 have been “glued” back on..................................................................................62

Figure 3.29 [Example]: A code comparison view for the $\text{NavigationPanel}$ class is provided to the developer. The difference between the two versions of the code is highlighted. The developer can confirm this change by clicking on OK or discard this change by clicking on Cancel if she finds any mistake...........................................63
Figure 3.30 [Example]: The popup notification is presented when a developer tries to modify any widgets that are dynamically generated.................................64

Figure 3.31 [Example]: Design/code hyper-linking brings the developer to the statement where the column labels are generated.........................................................64

Figure 3.32 [Example]: All the user-proposed requests are satisfied by using the dynamic editor. A key code “F1” is added for the “3 seconds” option, which is under the “Engine” menu. A new “Hint” button is added onto the navigation panel, and the labels of the columns are changed manually to support the descriptive notation.................................................................65

Figure 4.1 [Example]: A snapshot for the “Catalog Browser” from Java Pet Store 2. An accordion bar on the left is marked with dotted lines. An expanded accordion row, “Cats”, is marked with solid lines...............................................................79

Figure 4.2 [Example]: The function `setHeight` on its own lacks the calling context that is needed to properly associate the function with the accordion bar animation.........80

Figure 4.3 [Example]: Two function calls related to the accordion bar animation. The developer will need information to disambiguate the purpose of each function call. Some code is elided for illustration purposes.......................................................81

Figure 4.4 [Example]: The DOM mutator context of the `height` property mutator for the accordion row example, as described in Section 4.2.................................83

Figure 4.5 [Example]: The DMG of the accordion row’s execution.................................86

Figure 4.6 [Example]: Selecting a function call location from the calling context. The (?) entry references an anonymous JavaScript event handler function. A mutation of the style.height attribute for some DOM node was made in the function `setHeight` that is shown at the top of call stack........................................88

Figure 4.7 [Example]: A snapshot for the “Catalog Browser” from the JPS 2 Catalog Browser with markup to point out relevant interactive widgets described in the paragraphs of this section. An accordion bar on the left is marked with dotted lines. Label (A) is an expanded accordion row, “Cats”. Label (B) is a slider thumb, which is a thumb image for a particular pet. Label (C) is a scroll right button to roll the pets that are displayed in the scrollbar. Label (D) is a collapse button to
control the raising and lowering of the information pane. The “Information Pane” (E) has been revealed, and all information for this pet is presented to users. ..........89

Figure 4.8 [Example]: A front-end developer can use Firebug to easily find that height property is the key property of the accordion row animation. A developer positions the mouse cursor on the expanded accordion row “Cats” with Firebug’s inspect mode turned on, they will see the corresponding HTML elements that create the accordion row. When the developer puts the cursor onto an accordion row that is expanding/deflating, they will see the value of the height property is increased/decreased continually. In the current snapshot, the value of the height property is 120px, which reflects the open status of this accordion row............93

Figure 4.9 [Example]: The abstract behavior of an accordion row presented as a DMG. The two traces of the height values (overlayed on the model with block arrows, in the figure simply for illustration) show the information displayed to a developer when selecting one of the two nodes in the model.................................95

Figure 4.10 [Example]: Selecting a function call location from the calling context. The (?) entry references an anonymous JavaScript event handler function. A mutation of the style.height attribute for some DOM node was made in the function setHeight shown at the top of the call stack. This mutation corresponds to the height0 node from Figure 4.9. The stack contents serve to distinguish this execution of setHeight from those corresponding to node height1...........95

Figure 4.11 [Example]: A front-end developer uses Firebug to find the HTML element of the information pane widget. Using the inspection of Firebug, the developer is able to identify all the important properties of the information pane. When the developer positions the cursor on the information pane during the sliding up/down of this information pane, they will see the values of the identified properties increasing/decreasing respectively.................................................................98

Figure 4.12 [Example]: The code snippet from catalog.jsp. This code displays the HTML elements that create the information pane. Some code is elided for illustration purposes.........................................................................................................................99

Figure 4.13 [Example]: Step 1: the initial UI for the Catalog Browser without any user interactions (a). In this case, there are no trace values because the tracing is not
fired yet (b). Accordingly, the DMG is empty; only an arrow (1.1) is presented to indicate the start of the DMG (c).

Figure 4.14 [Example]: Step 2: the UI for the Catalog Browser when the Info Pane is fully expanded (a). During the expanding, my tool captures the various contexts in which attributes of the information pane and collapse button are mutated, and the trace information of value changes associated with each context are shown in the second column (some are elided for illustration) (b). The flow of the information pane and collapse button presented as a DMG is shown in (c). Each node corresponds to the entries from the table as shown in (b).

Figure 4.15 [Example]: Step 3: the UI for the Catalog Browser when the Info Pane is fully compressed (a). During the compressing, my tool captures all the newly executed DOM mutate contexts and associated values as shown in (b). Compared to Figure 4.14 (b), the newly captured DOM mutator contexts contain all the contexts that are executed to create the compressing animation. The corresponding DMG is evolved as shown in (c).

Figure 4.16 [Example]: Step 4: the UI for the Catalog Browser when the Info Pane is fully expanded again (a). The executed DOM mutator contexts are not changed because this info pane has been expanded before (b). The corresponding DMG is fully created (c).

Figure 5.1: The MVC architecture. Each box represents a component of MVC, for example, Model, Controller, or View.

Figure A.1: Screenshot from CrosswordSage program execution.
Figure A.2: Screenshot from Swing Designer for CrosswordSage class.
Figure A.3: Screenshot from FreezeFrame for CrosswordSage class.
Figure A.4: Screenshot from Java-Chess program execution.
Figure A.5: Screenshot from Swing Designer for Java-Chess class.
Figure A.6: Screenshot from FreezeFrame for Java-Chess class.
Figure A.7: Screenshot from JMSN program execution.
Figure A.8: Screenshot from Swing Designer for JMSN class.
Figure A.9: Screenshot from FreezeFrame for JMSN class.
Figure A.10: Screenshot from GanttProject program execution.
Figure A.11: Screenshot from Swing Designer for GanttProject class. .......................... 146
Figure A.12: Screenshot from FreezeFrame for GanttProject class. .............................. 146
Figure A.13: Screenshot from FreeMind program execution........................................... 147
Figure A.14: Screenshot from Swing Designer for FreeMind class................................. 148
Figure A.15: Screenshot from FreezeFrame for FreeMind class. ................................. 148
Glossary

AOSD: Aspect-Oriented Software Development
AST: Abstract Syntax Tree
BCM: Behavioral Concern Modeling
CSS: Cascading StyleSheet
CTG: Clustered Transition Graph
DMG: DOM Mutation Graph
DOM: Document Object Model
EFG: Event Flow Graph
EIG: Event Interaction Graph
FSM: Finite State Machine
GEF: Graphical Editing Framework
GUI: Graphical User Interface
J2EE: Java 2 Platform Enterprise Edition
JPS 2: Java Petstore 2.0
MVC: Model–View–Controller
NCLOC: non-commented lines of code
RIA: Rich Internet Application
TG: Transition Graph
UI: User Interface
UML: Unified Modeling Language
VFSM: Variable Finite State Machine
WHS: Widget Hierarchical Structured
Acknowledgements

I like to first thank the Computer Science Department of UBC for having given me the opportunity to participate, learn and grow with their PhD Program. The Software Practice Lab has been a home to me for the last 6 years. It is a faculty filled with great staff and colleagues. You have provided tremendous support, in guidance, knowledge and resources. Most of all, it will be the friendships and generosity that I have felt will forever be with me.

Most certainly, this has not been possible without the guidance of my supervisor, Eric. I thank you for taking me on as your student. You have shared much and I have learned much. Thank you for your patience and kindness.

To my Supervisory Committee, I am honored to have such a talented select group of people, Dr. Vuong and Dr. Mesbah. I thank you for all your advice. You all have made this possible for me and for that I am forever indebted to you.
Dedication

My parents are my greatest supporters and my foundation. They have always encouraged me to do my best and I hope that they are proud of my efforts. Love always.
Chapter 1

Introduction

GUI (Graphical User Interface) applications are widely used for processing user requests and performing user activities. In a GUI application, user interfaces are mainly used to interact with users by taking user input for operating the application and displaying the application’s output. The success of an application heavily relies on the appearance, preciseness, and correctness of the user interfaces.

User interfaces are pervasive and important in today’s software systems. As shown in published studies [76][75], in some software systems, graphical user interfaces can constitute as much as half the source code in typical projects.

As with any complex software development task, creating a user interface requires an iterative cycle of design and implementation. During software development or maintenance, the evolution of program features will require constant changes and upgrades to the GUI. Starting with an initial design, an interface is first prototyped and then refined over several cycles into a final product. At each stage, some design decisions may have to be reconsidered and the implementation adjusted accordingly. The GUI might even evolve after the release of an application in order to fix bugs or add new features.

After each cycle, developers can determine the quality of the current application by executing the implementation and evaluating GUI appearance and functionality. If they notice anything wrong with the appearance of the GUI, they have to map the problem back to some part of the implementation, to enact the appropriate change. Unfortunately, reversing engineering user interfaces for a software system and mapping the observed program behavior to the corresponding implementation can be quite difficult.
This difficulty is due to the fact that a developer working on the GUI might not have written the original code for all parts of the application. In that case, in order to understand how a user interface is implemented, the developer may have to dig through unfamiliar code to try and reverse-engineer the source or look for some assistance from associated documentation, e.g., requirements specification and code comments. However, this kind of documentation is often incomplete, outdated, or poor quality [69]. As described by Hassan et al. [42], “Currently, [code] inquiries can only be answered by scanning the source code for answers using tools such as grep, consulting documentation, or asking senior developers.” In this case, digging through the existing code to understand and maintain the implementation of user interfaces becomes the only choice for developers. In the next section, I further discuss the problems for understanding and maintaining GUI source code.

1.1 Problem Domain

Digging through existing GUI code to understand or change the existing implementation for some user interfaces is difficult. This difficulty stems from one complexity of software development: the relationship between output and source code is not always clear [44][102].

Typical software maintenance tasks require that programmers verify program correctness by examining program output (e.g., textual, graphical, and system I/O). Examination of output might be done manually or with the help of automated test cases [64][65][68]. When a problem is detected in the output, a typical strategy is to start examining the code at the point where the problematic output was generated.

In some cases, it could be cumbersome for programmers to map output back to the statements that generated the output. The reason is that the statements that generate output can be scattered throughout the implementation of several source modules. For example, typical logging frameworks and language pre-processors provide support to include both the name of the class and a line number where a logging statement was generated so developers can avoid these problems.

In my research, I am concerned specifically with the maintenance of GUIs. Developers who want to maintain the user interfaces of a GUI application have to fully understand the mapping relationship between a user interface and the corresponding implementation. This requires them to mentally create the mapping relationship between what they see on the screen and what is implemented in the source code. However, this mapping is often not easy to discover, because a user interface is normally generated by a large amount of code that is scattered throughout different software modules. Next,
two specific problems for understanding the UI-code mapping relationship are discussed in more detail.

- **Problem 1 - GUI Code Size Problem**: GUIs are key aspects in some software systems and can contain half the source code in some cases [75]. Therefore, the amount of GUI code in a software system can be very large. Scanning through the code and finding related implementation for some parts of a user interface can be quite cumbersome.

- **Problem 2 - GUI Code Scattering Problem**: the statements that generate a GUI can be scattered across the implementation of several source modules. The reason is that user interfaces are normally built by heavily reusing existing GUI code from different source modules. Understanding the implementation for some parts of a user interface require a developer to look through different source modules, which can be quite demanding.

### 1.1.1 Existing GUI Editing Tools

A popular approach to develop and maintain GUIs is to use WYSIWYG (What You See Is What You Get) editors, henceforth called GUI editors [2][15][86][91][45][103]. Typical GUI editors provide a round-trip (re-) engineering [78][77] process. A user interface created in the editor can be used to generate source. Later, the source code can be reverse-engineered statically to recover the user interface. For many years, GUI editors have attempted to provide a convenient model for developing and maintaining graphical user interface code through direct manipulation [13] of the interface elements. “Direct manipulation facilitates experimentation, has quick turnaround, and results in nicer-looking interfaces” [13].

GUI editors use a static view of the GUI, called a *design view*, to support a developer in two ways. First, by selecting widgets in a design view, the editor can display the source code location where a widget was created or some property of the widget was modified. This is useful when a developer wants to make some changes to the GUI manually in the source code. This kind of hyper-linking allows developers to seamlessly navigate between design view and source. I call this *view-based navigation*.

Second, by manipulating widgets in the design view, a developer can affect changes on the GUI. I call this *view-based editing*. These changes are translated by an editor into changes on the underlying source code. This allows the developer to reason visually about graphical changes. Changes that are
made from a design view are limited to “design-time” changes that are static in nature, because the visual nature of the editor lends itself to representing graphical relationships and not computations.

The view-based navigation feature and the view-based editing feature allow developers to use a design view as an entry point to guide their exploration of the source code of GUI implementation. This is a convenient approach to assist GUI developers in maintaining existing user interfaces. The reason is that the displayed GUI is always visible and easy to inspect visually compared to the source code implementation of the GUI.

### 1.1.2 Two Major Defects of Traditional GUI Editors

Traditional GUI editors are primarily used to prototype graphical user interfaces. Furthermore, the view-based navigation feature and the view-based editing feature can assist developers in addressing the two problems of maintaining GUI code. The reason is that developers who use GUI editors to maintain user interfaces can work on a design view to understand the large amount of GUI code that can be scattered throughout different source modules. However, traditional GUI editors have two major defects, which prevent them from assisting developers in solving the two problems.

- **Defect 1 - Limited Reverse-Engineering Capabilities:** the appearance of some parts of a user interface depends on the results of arbitrary computation and not on simple mapping between what developers see and what is implemented. In my studies on several large GUI applications, I found that the developers for applications first create GUI objects, called widgets. Next, these widgets are often assembled through dynamic computation code; for example, some widgets are arranged on a frame through a loop. However, the reverse-engineering capabilities of traditional GUI editors are severely limited and cannot properly interpret the GUI code that contains dynamic computation (e.g., loops or if-else). Therefore, traditional GUI editors often cannot completely reverse-engineer existing GUI code but can only provide an incomplete design view for developers. In Section 1.2.1, an example is provided to demonstrate this situation.

- **Defect 2 - No Explicit Support for Coordinated Behavior:** some parts of a UI are stateful and reactive. They could be transformed dynamically to different states based on a user’s interactions. Their appearance and behavior vary over time, based on mutations of states made from code, creating animation effects. In this case, if developers want to navigate directly to the code that is responsible for controlling the state transition of the UI, it could be
difficult for them to determine precisely the code that is responsible for controlling the UI transition and creating an effect or animation. Understanding the implementation for a coordinated UI behavior is difficult, and there is no existing GUI editor that can assist developers in understanding how a UI behavior is implemented. In Section 1.2.2, an example is provided to demonstrate this situation.

My research attempts to solve the above two defects for traditional GUI editors by focusing on the rendered on-screen interface. I refer to the dynamically rendered, runtime user interface as a *dynamic view*. A dynamic view can be interpreted by the programmer as one conceptual module (i.e., concern [81][94]) from the perspective of a software maintenance task. I say conceptual because the dynamic view may not map directly to one source code module. Still, by looking at a dynamic view, the programmer may see a cohesive set of elements (widgets) with a clear set of relationships. Relationships are formed from a hierarchical (tree-based), component-oriented model [29][58][93]. A dynamic view provides a complete, dynamic context from which to understand specific containment relationships. Mapping all these widgets and the relationships to source, however, requires understanding a scattered set of source elements. In my research, I seek to understand possible solutions to this software maintenance problem.

## 1.2 Research Objectives and Contributions

In this section, I provide further details of the two defects of traditional GUI editors. For each defect, I first present a motivating example to explain the problems that will be created for GUI developers by the defect. Then I describe my proposed solution to support my research objective. In Section 1.2.1 and Section 1.2.2, I describe defect 1 and defect 2, respectively. In Section 1.2.3, the contributions of my research are described.

### 1.2.1 Defect 1: Limited Reverse-Engineering Capabilities

#### 1.2.1.1 Motivating Example for Defect 1

Using a GUI editor, GUI developers can use a design view to guide them in understanding a user interface implementation that is normally spread across many different source modules. At first glance, it seems that the GUI Code Size Problem and the GUI Code Scattering Problem have been solved for non-animated user interfaces, because developers now work on a design view instead of a large amount of scattered GUI code. By using a GUI editor’s view-based navigation feature, a
developer can select widgets that they are interested in and then be navigated quickly to the source code that implements the widgets. By using a GUI editor’s view-based editing feature, a developer can modify the implementation of widgets that they are interested in through a WYSIWYG approach.

Unfortunately, the reverse-engineering capabilities of traditional GUI editors are severely limited. They use a static reverse-engineering approach to recover the design view and do not provide any support to deal with the dynamic computation code. In this case, traditional GUI editors fail to completely reverse-engineer the GUI code that contains both static code and dynamic computation code.

![JavaChess Screenshot](image)

**Figure 1.1 [Example]:** Screenshot from Java-Chess program execution. Sixteen classes contribute to this dynamic view.

In order to better demonstrate this defect, I used a user interface for a chess program as presented in Figure 1.1, in particular open source Java-Chess [31]. When the implementation of the Java-Chess user interface (Figure 1.1) is opened by an open source GUI editor, Swing Designer [91], the generated design view by the GUI editor is incomplete, as shown in Figure 1.2, because of defect 1.
The intuition behind this incomplete design view is based on an observation that much of the information in a dynamic view (as in Figure 1.1) consists of component sub-tree clusters that are static in nature. However, in practice, these sub-trees are often “glued” together with small pieces of dynamic computation. The approach used by traditional GUI editors cannot make sense of the dynamic code to find relationships between the static sub-trees, i.e., the analysis “gets stuck in the glue.” In the end, the entire tree-based view becomes truncated (as in Figure 1.2). Sub-trees are no longer visible in context because they are not reachable from the root widget in a design view.

![Figure 1.2: Screenshot from Swing Designer GUI editor for Java-Chess class. Several widgets are missing from the recovered design view. Extra rulers and grid markers are the effect of the editor tool in action.](image)

In large applications, traditional GUI editors cannot properly reverse-engineer existing GUI code, and they cannot provide complete design views because of defect 1. Therefore, traditional GUI editors are useful in partially solving the GUI Code Size Problem and GUI Code Scattering Problem for non-animated user interfaces. The assistance that they can provide is limited to all the widgets and relationships that can be interpreted statically and displayed. However, in 4 out of 5 open source applications that I looked at, the widgets and relationships that can be displayed by a traditional GUI
editor are no more than 50% of the entire user interfaces. I quantify these results as part of my evaluation in Section 3.6.

### 1.2.1.2 Proposed Solution

For traditional GUI editors, there is no solution for defect 1 (limited reverse-engineering capabilities) because of the limitations of the static program analysis. To address those limitations, I investigated using the dynamic analysis approach [9][85]. During the execution of a GUI program, user interface-related source code information is captured by instrumentation and then fed to a traditional GUI editor. Through this approach, the GUI editor will be able to provide a complete design view based on all the information that has been monitored dynamically. As discussed in Section 1.1.2, this new design view is a dynamic view rather than the static view that has been used by traditional GUI editors.

Both dynamic analysis and static analysis provide different tradeoffs. Whereas dynamic analysis is precise for a particular program execution, static analysis can explore many of the potential program paths in a program. In the next paragraph, I argue that the dynamic analysis approach can be well integrated with the existing GUI development procedure.

Recalling the GUI development procedure discussed at the beginning of this thesis, after each development iteration, GUI developers execute the application and browse different user interfaces they are working on. Finally, they see some problems or want to make changes for some widgets on a particular user interface. The next step is to map to the corresponding source code of the widgets that they are interested in. From this procedure, we can see that developers already execute the GUI application and trigger different user interfaces either manually or through some automated GUI test tools [64]. In this scenario, the developers are only interested in the precise and complete GUI information for a particular program execution path at a specific time. Therefore, executing GUI applications and using the dynamic analysis approach have the potential to be naturally embedded into developers’ development or maintenance work flow.

In my research, I use a dynamic view to replace the static view that has been used for traditional GUI editors. This dynamic view brings benefits for finding the UI-code mapping relationship, but it creates a problem for view-based editing: a developer using my tool now sees both static design information and dynamic information in the editor view. The static design information is editable. Editing the dynamic information from the editor is not supported, because it is only a computed result, although hyper-link navigation is still enabled.

For example, as shown in Figure 1.1, the columns are labeled from “a” to “h” on the bottom of the chess board. These labels are generated by the same code inside a loop, as shown in Figure 1.3. In the
traditional GUI editor, these labels are not visible, as illustrated in Figure 1.2. However, because my research allows a developer to see these labels, it has to prevent the developer from editing them because editing any individual label (e.g., changing the label “a” to “A”) on the design view will reflect to the changing of the code that is responsible for generating all the label objects. Therefore, all the label objects will be affected by this code change (e.g., all the labels turn to “A”), which is not desirable for the developer. In my research, I provide restrictions to prevent developers from editing the dynamic computed results, to avoid the above situation.

```
1. for(int i = 0; i < 8; i++) {
2.   char column = 'a' + i;
3.   JLabel label = new JLabel();
4.   label.setText(column+"");
5.   this.add(label);
6. }
```

Figure 1.3 [Example]: Java-Chess source code that is used to create the column panel of the chess board. Code has been refactored to improve readability.

1.2.2 Defect 2: No Explicit Support for Coordinated Behaviour

1.2.2.1 Motivating Example for Defect 2

As discussed, traditional GUI editors can be used to provide the mapping between widgets on the design view and the corresponding implementation in the source code. Because of the static nature of traditional GUI editors, this mapping cannot assist developers in understanding how a coordinated UI behavior or animation is implemented. This is an important defect for traditional GUI editors, because today’s UIs are very stateful and reactive, so this problem is of considerable concern for GUI development.

Here, I use a user interface from one rich interactive Web application called Java Pet Store 2 [87] as an example to explain defect 2 of traditional GUI editors. Then I motivate my solution for this defect.

In Java Pet Store 2, there is a rich interactive Web page called Catalog Browser, as shown in Figure 1.4. This page displays all the pet categories on the left column. When a particular pet category is selected, all pets in the category are displayed at the bottom row. Next, when a specific pet is selected, the corresponding large image and some descriptive information about the pet are loaded.
into this page. To motivate my research, I focus only on two widgets on this page, information pane and collapse button, which are highlighted both in Figure 1.4 and Figure 1.5.

The collapse button controls the sliding up and down of the information pane. It is an image element. There are two places in the code that set the source of the image element: the collapse icon will be changed to “up-to-down.gif” when the information pane is open (Figure 1.5) and “down-to-up.gif” when the information pane is closed (Figure 1.4). The information pane describes the detailed information for a selected pet. This widget is also mapped to an image element. When the pane is closed, it appears to slide behind the scrollbar (positioned beneath it). This information pane can be slid up and down when the collapse button is clicked, to show or hide the detailed information of a particular pet.

Figure 1.4 [Example]: A snapshot from the Catalog Browser page of Java Pet Store 2. The information pane is closed and only displays the name, rating, price and short description of the pet. The collapse button is an up arrow, to indicate that this button can be clicked to open this information pane.

In a GUI application, a UI behavior or animation varies over time, based on mutations of state made from code (e.g., JavaScript). The code mutates the properties of a widget to create a UI behavior. For example, the vertical sliding animation of the information pane is achieved by
continuously mutating several properties of the information pane widget, such as `clip` and `height` in the source code. The image-switching animation of the collapse button is achieved by setting different images for the collapse button at different times, based on whether the status of the information pane is closed or open.

Existing GUI editors such as Dreamweaver do not provide direct behavior-code mapping between a UI behavior and the actual implementation for the UI behavior. The reason is that the visual behavior of a widget (e.g., the way widgets are animated) is often achieved by a set of coordinated property-mutation statements. For example, the appearance of the collapse button may change to reflect that the button is an up arrow when the information pane is closed, and change again to reflect that it is a down arrow when the information pane is open. The changes to the appearance of the button and of the information pane have a causal relationship. The important point is that, if a developer wants to change the visual behavior, they may have to make coordinated changes to several statements.

Figure 1.5 [Example]: A snapshot from the Catalog Browser page of Java Pet Store 2. The information pane is fully opened and now displays the name, rating, price, short description, and full description of the pet. The collapse button is a down arrow, to indicate that this button can be clicked to close the information pane.
Currently, there is no existing GUI editor that can assist developers in leveraging their understanding of these causal relationships, seen in the browser view, in mapping from the browser view to source code. In this case, a developer who would like to make changes for a visual behavior has to face two problems: first, the developer has to find and memorize all the places where the related source statements are executed; second, the developer has to understand the coordinated relationship of all the related statements. In the next sub-subsection, I propose my solution based on the two problems.

1.2.2.2 Proposed Solution

Traditional GUI editors provide a simple mapping between what developers see and what has been implemented. However, to solve defect 2 of traditional GUI editors, there is no simple mapping between what developers see and what has been implemented. The reason is that what developers see is often a complex UI behavior. This behavior has to be mapped to different GUI statements, and all the statements work together in an appropriate temporal order to generate the visual behavior. Therefore, my research has to address two problems to solve defect 2 of traditional GUI editors: first, how to bridge the gap between a coordinated UI behavior and a set of GUI statements that create that behavior; and second, how to present visually the GUI statements to developers so they understand the coordinated relationship of all the statements.

Similar to the way defect 1 is solved, I also use the dynamic analysis approach to solve defect 2. In my research, a tool keeps track of the relationship between a UI behavior in the browser view and a set of statements in the code. My research prototype automatically builds a custom control-flow model for developers to understand the behavior-code mapping relationship. Therefore, a developer who would like to understand how a visual behavior is implemented can first interact with a widget of interest and then wait until the visual behavior is completed, e.g., a window is fully opened. Next, the developer is brought to a custom control flow model that records all the related source elements in an appropriate order. This model uses nodes and edges to present the abstract high-level pattern of the visual UI behavior. Using this model, developers can focus quickly on the coordinated relationship of the source elements that they are interested in. Next, they can use the obtained behavior-code mapping to understand or modify the implementation of the UI behavior.

This custom control flow model provides a high-level abstraction for developers to understand a particular visual behavior. As discussed in Section 1.1.2, it is difficult to determine precisely the code that is responsible for controlling the UI transition between states and creating a particular effect. In the actual implementation of a visual behavior, each UI state is transitioned to another state through the execution of one GUI statement or several. By identifying all the related GUI statements for each
UI state transition and presenting them in a high-level abstract model, developers can easily use the model to map a visual behavior to all the related GUI statements that are responsible for generating the behavior.

### 1.2.3 Contributions

In this thesis, I investigate dynamic approaches for improving view-based maintenance of traditional GUI editors for non-animated and animated user interfaces. Traditional GUI editors have two major defects: limited reverse-engineering capabilities and no explicit support for coordinated behavior. To address these two defects, I use dynamic monitoring approaches to improve the understanding of existing GUI designs by providing a dynamic context for making informed maintenance decisions. The contributions of my research are presented as following:

- Define and describe the two defects of traditional GUI editors in Chapter 1.

- Implement and evaluate my solutions for the two defects in Chapter 3 and Chapter 4.

- Formally define the code analyzing algorithms that are used by traditional GUI editors in Chapter 3.

The rest of the thesis is structured as follows: Chapter 2 provides the background of this thesis, Chapter 3 addresses the first defect, Chapter 4 addresses the second defect, Chapter 5 presents related work and I conclude in Chapter 6.
Chapter 2

Background

This chapter presents some background knowledge that is used in this thesis. GUI components such as buttons and frames are called widgets, and I describe the unique characteristics of these components in Section 2.1. Next, I briefly introduce GUI editors (Section 2.2) and describe the process for engineering user interfaces using a traditional GUI editor. Then, I introduce GUI events and event handlers in Section 2.3. In Section 2.4, I describe the responsibilities of GUI developers. Finally, I describe AOSD (Aspect Oriented Software Development) I used in my research (Section 2.5).

2.1 Structured GUIs, Widgets and GUI Design

In this research, I am only interested in the GUIs that are well structured. In this case, GUIs can be broken into different components, and each can be mapped to its implementation. Therefore, a developer will be able to use each component that appears on the output as an “anchor” to explore its implementation. In my research, I call this kind of GUI as a structured GUI. For example, a user interface developer most likely uses GUI components in a Java GUI toolkit, for example, Swing [89] or AWT [8], to implement a user interface instead of using raster graphics to draw all the pixels of the user interface. In my research, I chose two kinds of GUI applications that have structured GUls: Swing-based (i.e., Java) desktop applications and Document Object Model (DOM) [100]¹ based Web

¹ DOM is a language-independent, cross-platform, tree-structured representation of the entire Web page.
applications. Next, I explain two important concepts of structured GUI outputs, widgets and widget hierarchy trees.

Object-oriented GUIs make use of objects called widgets [1]. A widget is an object created from a specific set of classes provided by a GUI framework or a custom subclass of those classes. I call these classes widget classes. Widget classes can be further broken down into framework widget classes and custom widget classes. Framework widget classes, for example, JButton and JFrame, are from any GUI framework libraries (e.g., Swing or AWT). I call widget classes that are custom, application specific and subclasses of framework widget classes custom widget classes. For example, a custom widget class called AlphaColorJButton extends a JButton framework widget class from Swing library and enhances the color feature for the JButton class. GUI editors, much like component-based programming and dynamic languages (e.g. Smalltalk [50]), often blur the line between compile-time and runtime, so I am careful to use widget to refer to an object and widget class to refer to a class.

Soon after Web applications became conventional, the widget concept was introduced to Web UI developers. In a Web application, HTML, XML and CSS are used to create widgets. For example, a button in a Web application can be declared by using a <button> HTML tag. During runtime, a button object (widget) is created.

A widget hierarchy tree is a tree of widgets. It is used to represent the hierarchical relationship for all the widgets in a particular user interface. This tree model is often used internally by GUI editors.

A complete GUI design consists of both static design and dynamic design. Developers create the static design of a user interface without using any dynamic computation such as loops and conditionals. The dynamic design of a user interface is only a computed result. For example, a panel widget displays different titles based on some dynamic information such as a user’s login status. In this case, the size of the panel is a static design but the title of the panel is a dynamic design. The static design and dynamic design of a user interface work in unison and can be tangled together in the user interface implementation. Ensuring a complete separation of the user interface static design code and the dynamic design code would require significant effort from developers. Developers may also have prioritized decomposition decisions along other dimensions of concern, which conflict with this separation [94].

### 2.2 GUI Editors

WYSIWYG GUI editors have been used widely for developing and maintaining user interfaces for both desktop applications and Web applications, for example, Visual Editor [45] and Swing Designer.

2.2.1 Forward Engineering

The forward engineering of a GUI editor begins with the developer who envisions the design of the graphical user interface. A developer works directly with the design view of an editor by manipulating widgets (a). As changes are made to the design view, the editor maintains the design model (b). This design model is a widget hierarchy tree model. Each change to the design model can add a new widget somewhere in the hierarchical model, remove a widget, or change a property of a widget. Finally, the editor realizes the implementation of the GUI by generating code from the corresponding design model (c).

![Figure 2.1: Forward- and reverse-engineering process for graphical user interface editors.](image)

2.2.2 Reverse-Engineering

GUI editors provide reverse-engineering by allowing a user to select a custom widget class (c) that they want to edit. For example, to provide Figure 1.2, I chose the custom class “JavaChess” from the Java-Chess project. The editor analyzes the class and recovers a design model (b) to display in the view (a). Next, developers can use this design view as a guideline to understand and maintain different widgets and their relationships on the user interface.

Most GUI editors use a static analysis approach to reverse-engineer existing user interface code and build design models. Most of the time, a design model is incomplete because some widgets or
relationships could not be recovered by static analysis. Furthermore, the corresponding design view is incomplete, and it is just a crude estimation of how the user interface will look at runtime. Therefore, the recovered design view can only provide limited design information for developers. In Chapter 3, I further discuss this problem and provide a possible solution.

2.3 GUI Events and Event Handlers

An important feature of a GUI program is to interact with users. When a user acts on a widget, for example, clicking on a button or opening a window, the GUI program needs to recognize the “clicking” and “opening” events. It then needs to respond to the events with some actions. GUI events are important concepts in GUI programming, and they are defined by GUI libraries. Every widget that has been defined by GUI libraries has some associated predefined events. Those events specify what “signals” the widget responds to. For example, a Swing JButton widget can respond to a click event but cannot respond to a drag and drop event.

When an event happens, a corresponding response has to be provided by the GUI program to respond to the user’s action. This response is programmed by GUI developers, and they write an event handler function to respond to the event. This event handler function is only executed when a specified event is fired. When the event handler function is executed, the event and some associated arguments are passed into the event handler function. Therefore, developers can make an appropriate response for the fired event. In Chapter 4, my analysis is focused on particular GUI events and their corresponding event handler functions.

2.4 GUI Developers

Creating good graphical user interfaces requires considering the appearance and careful design for usability, and providing for concrete implementation. Decent graphical user interfaces require cooperation from both GUI designers and GUI programmers, who are all called GUI developers. GUI designers are creatively driven and are focused on the appearance, style, and layout of the user interfaces. They are not familiar with writing code and are not involved in the task. After GUI designers design the appearance of the user interfaces, their design is given to GUI programmers who will implement all the user interfaces.

However, many companies mix the job responsibilities of GUI designers and GUI programmers. In this case, my research is valuable because GUI developers expect some simple and convenient way
to create and maintain graphical user interfaces, especially through a WYSIWYG approach. GUI developers sometimes are also called *front-end developers.*

### 2.5 AOSD (Aspect-Oriented Software Development) and AspectJ

Program concerns such as user interfaces, security, and logging are spread throughout the program in an undisciplined way [95]. AOSD was introduced to deal with this crosscutting concern by using a unit of modularity called *aspect.* AOSD allows developers to augment code in selected places of a program to monitor a particular crosscutting concern at runtime. AspectJ [53][52] is a Java implementation for AOSD. In AOSD, the places where a developer can augment code are called join points. A *pointcut* is used to specify particular join points to augment code. For example, a pointcut `call(javax.swing.*+.new(..))` intercepts all the creation of widget classes. When this pointcut matches a join point, an *advice* is executed, and the reflective capabilities of AOSD can be used to record information for argument values and corresponding source locations (i.e., classes and line numbers). Finally, an aspect in AOSD is a set of pointcuts and advice. AOSD provides a modularized way for developers to enhance Java code. In Chapter 3, AspectJ is used to solve defect 1 of traditional GUI editors.
Chapter 3

Dynamic Round-Trip GUI Maintenance

As discussed in Chapter 1, it is difficult for developers to understand and maintain GUI applications because of the GUI Code Size Problem and GUI Code Scattering Problem. WYSIWYG GUI editors could be used to try to solve these problems for non-animated GUIs. However, due to defect 1 of traditional GUI editors, the reverse-engineering capability of traditional GUI editors is severely limited, and the design views that are reverse-engineered by traditional GUI editors are often incomplete or sometimes truncated. These incomplete or truncated design views can only provide very limited assistance for GUI developers to work with. This limited assistance is demonstrated from the two aspects of GUI editors’ functionalities: view-based navigation and view-based editing.

The limitation of view-based navigation is mainly derived from the poor reverse-engineering capability of traditional GUI editors, which are strongly restricted by the fact that the static program analysis cannot completely reverse-engineer existing GUI code. Most of the design views that are recovered by traditional GUI editors can only provide incomplete user interfaces compared to the actual runtime user interfaces. For the five open-source GUI applications that I have investigated, a traditional GUI editor can only provide one reasonable reverse-engineered design view that GUI developers can use to maintain and understand existing GUI code. For the other four GUI applications, GUI developers lose the view-based navigation feature for all the widgets that cannot be rendered by the traditional GUI editor.

The limitation of view-based editing is due to incomplete design views. The widgets that cannot be rendered on incomplete design views are not visible and cannot be edited by developers who use traditional GUI editors. As discovered in my investigation, the number of widgets that can be
displayed is quite small; therefore, the view-based editing feature of traditional GUI editors is also restricted by the traditional GUI editors’ poor reverse-engineering capability.

The rest of this chapter is structured as follows: Section 3.1 motivates the need for my research by demonstrating how a GUI developer tries to solve some code evolution tasks through a traditional GUI editor. In Section 3.2, I explain in detail the internal code analyzing technique and defect 1 of traditional GUI editors. Section 3.3 presents view-based navigation. Section 3.4 presents view-based editing. In Section 3.5, I explain how to solve the same code evolution tasks that are presented in Section 3.1, by using my research. Section 3.6 presents a quantitative evaluation.

3.1 Motivating Examples

![Image of Java-Chess](image)

Figure 3.1 [Example]: The dynamic view for the main window of the Java-Chess program. The intended editing tasks are marked by: a) the “3 seconds” menu item, b) a button panel, and c) the column labels of the chessboard.

To motivate my research, I use a chess application called Java-Chess [31]. In this section, I present three motivating examples to illustrate how to modify a user interface of an existing desktop GUI
application. In these examples, a developer tries to get assistance from a traditional GUI editor, Swing Designer.

Consider a scenario in which a developer wants to perform maintenance on the user interface for the Java-Chess program, in particular the main window of Java-Chess. The developer may first need to plan a change to make, by reasoning about some dynamic view, as shown in Figure 3.1.

In this motivating example, the developer has just been assigned three tasks to extend the Java-Chess program according to some users’ requests.

- (a) Some users complained that it is inconvenient not to have a keyboard shortcut for an often-used menu item. So, they have to choose from the menu “Engine → Search Time → Fixed time → 3 Seconds” every time. In this case, they need a shortcut for that menu item.

- (b) Some users said that they need a “Hint” button on the navigation panel to give them advice when they are stuck.

- (c) Some users would like to use a traditional chess board notation instead of the abbreviated algebraic notation for column labels. Accordingly, they hope that the developer can change the labels of the columns of the chessboard from “a-h” to “QR-KR” for their convenience.

In order to implement those users’ requests, the developer may first need to plan the changes by reasoning about some dynamic view, as shown in Figure 3.1. In this scenario, the developer would like to a) add a key code “F1” for the “3 seconds” option that is under the Engine menu, b) add a new “Hint” button under the chess board, and c) change the labels of the column from “a-h” to “QR-KR”.

Dynamic views are a product of program execution so they are not directly available for editing by the developer. So, when the developer has an intended change in mind, she would have to consider how to implement the change. Two possibilities for implementation are to make changes to the source directly or use the help of a GUI editor. I consider each of these in turn.

First, to change the source directly, the developer would have to know the method call source element that generated the widget (or widget property) that is to be changed. These calls could take some time to find, since the source code that is used to generate a dynamic view may be scattered across the source. In Figure 3.1 there are 16 classes that contribute to the visual appearance that
includes GUI code tangled with program logic. The developer would have to remember how the statements of code in all these classes related to the dynamic view or else reconstruct all these relationships.

A second option for the developer is to use a GUI editor. It would seem reasonable that the developer should be able to simply click on some part of the view and ask to be taken to the source element where that part of the view was generated. Also, for parts of the view that are static design, the developer should be able to edit directly from the view. This is possible in very limited cases with existing GUI editors as I discuss next.

![Swing Designer GUI editor for Java-Chess class](image)

**Figure 3.2 [Example]:** Screenshot from Swing Designer GUI editor for Java-Chess class. Several widgets are missing from the recovered design view. Extra rulers and grid markers are the effect of the editor tool in action.

Current GUI editors are crippled once the initial GUI design code has been integrated with the main program logic. As an example, consider the same view rendered by a GUI editor in Figure 3.2. We can see that several of the widgets are missing: two menubar menus, the entire chessboard, and the game player information panel (in the upper right quadrant). The particular GUI editor to produce Figure 3.2, Swing Designer, was rated as the most complete Eclipse IDE-based editor by an online article [86]. This is unfortunate considering that much of the information missing is part of the static
design. As an example, the missing menu items and associated drop-down menus never change during the course of Java-Chess execution.

This scenario is used to demonstrate that mapping intended changes from the dynamic view to the implementation is difficult to do manually and not possible for complete programs using existing editors.

Now that we understand all the problems for maintaining GUI code by using the two existing approaches (i.e., changing code directly or using a traditional GUI editor), let us look into the three concrete tasks that have been assigned to the developer. In the remaining part of this section, I will show how the developer tries to modify some existing widgets for the three tasks by using a traditional GUI editor. When the developer has the intended changes in mind, she first has to find the source files (class files) that correspond with the widgets. After researching the source, the developer may be able to find the classes that she needs to change.

![Figure 3.3 Example: Adding a new “Hint” button in the developer’s intended position of the NavigationPanel.](image)

For task (a), the developer searches for the text “seconds” to find the class where task (a) can be completed. Luckily, this class is easy to find by simple text search. This brings her to the class ChessEngineImpl in the javaChess.engine package. Next, when ChessEngineImpl is
opened by the traditional GUI editor for task (a), unfortunately the Engine menu cannot be displayed because of the limited reverse-engineering capabilities of the traditional GUI editor.

For task (b), there is no text cue available to help the developer locate the proper class so she examines the javaChess.engine package further. Unfortunately the developer cannot find any classes that are related to task (b) in this package. So now there are 87 other classes in 23 packages to explore to find the correct class. This could be difficult even for this small program of only 5,600 NCLOC. Eventually, the developer finds the NavigationPanel for task (b). Finally, when the developer opens the NavigationPanel she will be able to see the navigation panel as expected, as shown in Figure 3.2. However, when she edits the navigation panel, she only sees the navigation panel and loses the context of the whole main window. In the next paragraph, this example is used to demonstrate that sometimes a context is useful for developers to make decisions about potential changes.

**Figure 3.4 [Example]:** The added new “Hint” button is squeezed to the right side of the navigation panel at run time because the developer did not consider the context of the panel’s container.
For this task (b), the developer adds a new “Hint” button on the left side of these existing four buttons, as shown in Figure 3.3. Since the “Hint” button has a different function than the navigation buttons, the developer decides to put a gap between the “Hint” button and the four navigation buttons to distinguish them. However, later on, when she sees the modified navigation panel in the executed main window as shown in Figure 3.4, she finds that the added button is not in the position she expected; all the buttons are huddled together. The reason is that the container of the navigation panel specifies constraints on the layout. So, when the panel is added into the container, it is restricted and the added button is squeezed to some unexpected position. This unsuccessful editing is due to the developer not being able to complete the task in the global context of the dynamically generated main window with the traditional GUI editor. In the end, the developer loses the expected WYSIWYG feature for GUI editing.

![Figure 3.5](image)

**Figure 3.5 [Example]:** The ChessBoardRenderer2D class is opened by using the GUI editor. All the labels for the columns and rows cannot be rendered because of the limited reverse-engineering capabilities of the GUI editor.

Task (c) is similar to task (b): the developer also has to explore all the existing classes in all the packages of the Java-Chess project to locate the ChessBoardRenderer2D. Unfortunately, when
ChessBoardRenderer2D is opened, the labels on the chessboard cannot be displayed as shown in Figure 3.5, because of the limited reverse-engineering capabilities of the GUI editor.

In my research, I seek to address these kinds of problems by allowing developers to interact directly with a dynamic view. Developers execute the program they want to edit. A dynamic analysis creates a mapping between the dynamic view and the source code, and static checking prevents changes to elements that are not part of the static design. Before presenting my solution, I further explore defect 1 of traditional GUI editors in the next section.

### 3.2 Defect 1: Limited Reverse-Engineering Capabilities

GUI editors are widely used to create and manage user interface implementation that is written by using GUI framework libraries. Using a GUI editor is convenient for programming GUI applications because it provides some useful assistance in GUI programming, for example, round-trip GUI editing, drag-drop GUI programming, and reverse-engineering existing GUI code.

In this section, I am not going to discuss further the convenient features of traditional GUI editors. Instead, I attempt to find the reasons why existing GUI code often cannot be reverse-engineered and rendered properly.

Commonly, most traditional GUI editors are only useful in analyzing the code that has been generated by the GUI editors themselves. Most of the hand-written GUI code cannot be analyzed and rendered properly by traditional GUI editors. Therefore, if a developer wants to use a traditional GUI editor for the hand-written code, they have to do code refactoring on the existing code to make the code fit the GUI editor’s analysis. Sometimes this code refactoring is not straightforward, and it requires full understanding of the GUI code. However, fully understanding and refactoring existing GUI code would suffer from the two GUI understanding/maintenance problems discussed in Chapter 1.

In the rest of this section, I first define a clear research domain for all the concepts and programming language terms that are important to GUI editors (Section 3.2.1). Next, I present the analyzing and rendering techniques that are used by traditional GUI editors (Section 3.2.2). Finally, I explain defect 1 of traditional GUI editors in detail (Section 3.2.3).

#### 3.2.1 Terminology Domain

Table 3.1 lists a set of Java terms, which are the subset of the Java languages specification [41]. This list contains all the Java terms I discuss in this chapter. The GUI editor’s features are primarily
centered on concepts related to these terms. Therefore, I set these Java terms as the terminology domain to discuss. In the second column of Table 3.1, I describe how each term relates to the GUI domain, which is important to GUI editors.

Table 3.1: The subset of the Java language specification contains all Java terms that are important to my research. The Java terms are listed in the first column of this table. The second column describes the understanding of these terms from the GUI domain.

<table>
<thead>
<tr>
<th>The subset of the Java Language Specification</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>GUI editors are mainly focused on visual classes. There are two types of visual classes: widget classes (framework widget classes and custom widget classes(^2)) and any classes that contain the entry point to the creation of a GUI, for example, the main method.</td>
</tr>
<tr>
<td>Object</td>
<td>A widget is an object created from a widget class. If a widget is created from a framework widget class, then it is called a framework widget; if a widget is created from a custom widget class, then it is called a custom widget.</td>
</tr>
<tr>
<td>Constructors</td>
<td>The constructors of custom widget classes.</td>
</tr>
<tr>
<td>Methods</td>
<td>All the methods in custom widget classes.</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Inheritance is widely used in GUI programming to create widget classes. All GUI framework libraries are built based on inheritance instead of creating each new widget class from scratch. For example, in the Swing GUI framework library, the MetalComboBoxButton widget class is extended from another widget class, JButton.</td>
</tr>
</tbody>
</table>

---

\(^2\) Framework widget classes and custom widget classes are defined in Chapter 2.
<table>
<thead>
<tr>
<th>The subset of the Java Language Specification</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements</td>
<td>GUI editors are focused on any statements that instantiate widget classes to create widgets or make use of widgets. I call these statements <em>GUI statements</em>. There are three important kinds of GUI statements: <em>widget instantiating statement</em>, <em>widget property-setting statement</em>, and <em>hierarchy relationship statement</em>.</td>
</tr>
<tr>
<td>Object Creation Statements</td>
<td>Widget instantiating statements are used to create widgets by either instantiating framework widget classes or instantiating custom widget classes, e.g., <code>JButton button = new JButton();</code>. Widget instantiating statements can be further broken down to <em>framework widget instantiating statements</em> and <em>custom widget instantiating statements</em> based on the kinds of widget classes that the statements instantiate.</td>
</tr>
<tr>
<td>Method Invocation Statements</td>
<td>Widget property-setting statements are used to set the properties of particular widgets, e.g., <code>jbutton. setText(“button”);</code>. Hierarchy relationship statements are used to build the tree structured relationship for user interfaces, e.g., <code>jframe.add(jbutton);</code>.</td>
</tr>
<tr>
<td>Control flow statements</td>
<td>Control flow statements can be used to control the generation of user interfaces. For example, the decision-making statements (if-then, if-then-else, switch) can be used to create different login UIs for different users. The looping statements (for, while, do-while) can be used to create more complex UIs instead of manually creating each individual piece.</td>
</tr>
</tbody>
</table>
Now that we have a clear understanding of all the Java terms that are important to my research, I can explain the internal analysis mechanisms that are used by traditional GUI editors, in the next subsection.

3.2.2 GUI Editor Code Analysis

Here, I present the analyzing and rendering techniques that are used by Swing Designer. Other GUI editors use similar techniques but with only minor differences. Therefore, I use a more generic name, “traditional GUI editor” (or GUI editor), instead of Swing Designer, in the rest of this subsection.

A GUI editor is mainly used to assist GUI developers in visually editing a visual class. To render a visual class, a GUI editor has to first analyze and reverse-engineer the class. During this reverse-engineering process, a design model is statically recovered using information in the constructors or the main method of the visual class.\(^3\) This is where a large part of the complexity of GUI editors lies, so I describe the details in the next three parts. To recover a design model, GUI editors must first identify a root method (Section 3.2.2.1) and then analyze the root method to build a design model (Section 3.2.2.3). In Section 3.2.2.2, I describe the semantic domain of GUI editor analysis.

3.2.2.1 Identifying a Root Method

A GUI developer first chooses a class file to open by using a GUI editor. This opened class file is called *opened class*. The GUI editor starts by identifying the entry point of the analysis, which is called the *root method* (line 21 of Figure 3.6) of the opened class (line 1 of Figure 3.6).

If the current opened class is a custom widget class (line 3 of Figure 3.6), and the custom widget class only has one constructor, then this constructor is identified as the root method (line 13 of Figure 3.6). However, if the custom widget class has more than one constructor (line 4 of Figure 3.6), the editor first tries to identify a constructor with zero arguments and then set this constructor as the root method (line 6 and line 7 of Figure 3.6). If the editor cannot find a constructor with zero arguments, the editor is not able to decide which constructor to use as a root method. In this case, a GUI developer has to assist the editor in finding an appropriate root method by adding Java annotations in the source code (line 8 and line 9 of Figure 3.6).

In another scenario, if the opened class is not a custom widget class, the editor must try to find a *main* method and set this method as the root method (line 15 and line 16 of Figure 3.6). If this

---

\(^3\) As well as methods called during constructor execution.
main method does not exist, an exception will be thrown to indicate that a developer is trying to open and analyze a non-visual class (line 18 of Figure 3.6).

```plaintext
1. Input: openedClass
2. BEGIN
3. IF openedClass ∈ {custom widget classes} THEN
4.    IF |openedClass.constructors| > 1 THEN
5.       FOR each constructor of openedClass.constructors
6.          IF |constructor.argument|=0 THEN
7.             rootMethod := constructor
8.          ELSE IF constructor.isAnnotated THEN
9.             rootMethod := constructor
10.        END IF
11.       END FOR
12.    ELSE
13.        rootMethod := openedClass.constructor
14.    END IF
15. ELSE IF openedClass.isContain(a main method) THEN
16.    rootMethod := the main method
17. ELSE
18.    THROW exception
19. END IF
20. END
21. Output: rootMethod
```

Figure 3.6: The algorithm used to identify the root method of a given visual class.

### 3.2.2.2 Semantic Domain of GUI Editor Analysis

After the root method is identified, the next step is to analyze it and then build a design model. Before this analysis procedure is described, I first explain the semantic domain of GUI editor analysis and then provide the procedure for how a GUI editor builds a design model in the next sub-subsection. An important concept in the semantic domain of GUI editor analysis is a GUI constant. I first provide the definition of GUI constant.
**Def 3.1: GUI constant**

GUI constants are a superset of the language constants such as string literals and integer constants. This set also includes statements and expressions that make reference only to classes and methods in the GUI framework libraries, as well as certain classes and methods in the core language libraries.

After identifying a root method, a GUI editor starts to analyze statically each statement in the root method. During the analysis, any statements that use variables that cannot be determined statically to refer to a single GUI constant are removed from consideration. This determination is limited to the analysis of the constructor, methods called during constructor execution, the main method, and any fields that include initializers. Therefore, traditional compiler (intra-procedural) analysis is sufficient to make a conservative approximation.

Most traditional GUI editors use intra-procedural analysis. Therefore, they are flexible enough to resolve simple intra-procedural data-flow relationships between variables and the values they reference. For example, typical Java editors would understand `new JButton("OK")` in the same way even if the string literal “OK” was stored in an immutable (i.e., final) field in the same class. GUI editors are also hard-wired to understand the use of specific localization resources for storing and retrieving strings in different languages. For example, typical Java GUI editors would understand the expression `new JButton("OK", getResource("button.gif"))` without any need to execute this code.

For building the model, traditional GUI editors must distinguish framework widget instantiating statements from custom widget instantiating statements because editors have hard-wired knowledge for framework widget classes but not for custom widget classes.

Traditional GUI editors understand the semantics of framework widgets without any analysis of framework code. Knowledge of framework widgets is hard-wired into the editor itself. Essentially, during static design model recovery, the editor is able to treat framework widgets as if they were program constants (i.e., string literals, number literals, etc.). For example, a call to “`JFrame frame = new JFrame()`” is considered a statically resolvable constant, so interpreting this statement allows the editor to include a JFrame widget in the static design model. Here, I say that framework widget instantiating statements can be *statically interpreted.*

---

4 The analysis of GUI editors are only limited to the methods that are called on the implicit this object. More details are explained later in this section.
5 This excludes public static fields that could have been set prior to constructor execution.
6 Some editors even construct data-flow relationships that are unsound (i.e., might not hold for all executions), in order to promote flexibility. In my research, I assume only sound inferences are used.
7 `getResource()` is a Java idiom for retrieving resources such as images and internationalization strings from disk.
However, traditional GUI editors have no knowledge to statically interpret custom widget instantiating statements because GUI editors do not have hard-wired information for custom widget classes; for example, a GUI editor is able to understand how to interpret “new JButton ()” but does not understand the instantiation of a custom button class, e.g. “new AlphaColorJButton()”. In this case, during the analysis of given classes, GUI editors have to instantiate (actually execute) the encountered custom widget instantiating statements through separate virtual machines. The reason is that traditional GUI editors always try to display as many widgets as possible for developers. In this case, developers can obtain a better estimation of the runtime appearance of the existing GUI code. Here I say that custom widget instantiating statements can be dynamically instantiated.

3.2.2.3 Building a Design Model

Recalling the design model that has been described in Chapter 1, a user interface can be considered as a component tree that has a hierarchical relationship among all of the widgets. Here, I describe how this model is built.

During the code analysis, any statements nested in loops and conditionals are removed from consideration, including loops or conditionals (line 4 and line 5 of Figure 3.7). If the encountered statement is a framework widget instantiating statement (line 8 of Figure 3.7), this statement can be statically interpreted (line 9 of Figure 3.7) by the GUI editor; if the encountered statement is a custom widget instantiating statement (line 15 of Figure 3.7), the GUI editor attempts to instantiate the custom widget class (line 16 of Figure 3.7). After the widgets are statically interpreted or dynamically instantiated, the model is updated with the widgets (line 10 and line 17 of Figure 3.7). During the code analysis, any of the encountered hierarchy relationship statements such as adding one child widget onto its container widget will be statically interpreted (line 12 of Figure 3.7) because the GUI editor is able to understand these statements. Furthermore, the model is updated with the interpreted hierarchy relationships (line 13 of Figure 3.7).

During the analysis of the root method, if the GUI editor detects any calls to other methods and the receivers of the call sites are implicit this objects (line 18 of Figure 3.7), then those methods will be statically analyzed by the GUI editor as it encounters the method calls (line 19 of Figure 3.7). Similarly, if those methods call other methods that are called on implicit this objects then those methods will be analyzed recursively [16]. If the GUI editor finishes analyzing the root method and all the methods called by the root method, then the GUI editor will stop analyzing the code.

---

8 This also includes any code that might execute after a return statement controlled by conditionals and by recursive methods. I do not consider exceptional control-flow or concurrency in my implementation.
1. Input: rootMethod
2. BEGIN
3. FOR each statement of rootMethod
4. IF statement ∈ \{Control flow statements\} THEN
5. Do nothing
6. ELSE
7. IF statement ∈ \{GUI constants\} THEN
8. IF statement ∈ \{framework widget instantiating statements\} THEN
9. widget := staticInterpret (statement)
10. widgetSet := widgetSet U \{widget\}
11. ELSE IF statement ∈ \{hierarchy relationship statements\} THEN
12. relationship := staticInterpret (statement)
13. relationshipSet := relationshipSet U \{relationship\}
14. END IF
15. ELSE IF statement ∈ \{custom widget instantiating statements\} THEN
16. widget := dynamicInstantiate (statement)
17. widgetSet := widgetSet U \{widget\}
18. ELSE IF statement ∈ \{method invocation statements with implicit this receivers\} THEN
19. RECURSIVE CALL (method)
20. ELSE
21. Do nothing
22. END IF
23. END IF
24. END FOR
25. END
26. Output: widgtSet, relationshipSet

Figure 3.7: Analyzing the identified root method and generating one set that contains all the identified widgets (widgtSet) and another set that contains all the identified widget hierarchy relationships (relationshipSet).
As we can see, this recursive procedure is only applied to the methods that are called on implicit this objects. The GUI editor will skip any method calls that are called on any other objects. The reason is that the GUI editor restricts its analysis to the methods for which the target is easily resolvable.9

The above code analysis procedure10 supposes that all the widget instantiating statements, hierarchy relationship statements, and method invocation statements do not take any parameters. If any parameters are required to interpret or instantiate the statements, GUI editors will use a simple intra-method dataflow analysis to resolve the actual values of the parameters and then pass the values to the statements. In this case, any unresolved parameters will result in the widgets or hierarchy relationships not being successfully interpreted or instantiated, which creates an incomplete design model. Different GUI editors use different policies to handle this problem. More details are discussed in Section 3.2.3. Therefore, in the algorithm description as shown in Figure 3.7, the details have not been described.

After analyzing the given GUI code by using the algorithm as presented in Figure 3.7, the widgets and the hierarchy relationships among the widgets are obtained. Next, a design model can be created easily by using a simple tree-building approach described in [84]. After the design model is created, this model is often displayed to GUI developers through GEF (Graphical Editing Framework) [40].

Figure 3.8 presents a code example to illustrate the code analysis technique described above. When the demoEditor class is opened by using a GUI editor, for example, Swing Designer, the editor first finds the entry point of this class (root method), i.e., the main method, and then the GUI editor starts analyzing this main method. During the code analyzing of the main method, the GUI editor statically interprets line 5 to line 10 because the information, such as JFrame creation, layout setup, and string type, have been hard-coded into the GUI editor. (In Table 3.2, I list a set of types and methods that have been hard-coded into the GUI editor for the code example in Figure 3.8). By understanding those five lines, the GUI editor creates a partial design model that only contains the widget frame and the layout information. Next, the GUI editor attempts to analyze line 12 and line 13, but it has no built-in knowledge for the TextJButton class. Because these two custom widget instantiating statements take parameters, the GUI editor will first try to figure out the actual values for the variables button1 and button2 by using a simple intra-procedure data-flow analysis.

9 In general, statically determining the target function for a polymorphic method call is quite difficult [86].

10 To simplify the algorithm, I did not discuss how the algorithm processes widget property-setting statements. For each encountered widget property-setting statement, the GUI editor interprets the statement and then sets the property of the corresponding widget.
Figure 3.8 [Example]: This code snippet is used to illustrate the code analyzing technique used by traditional GUI editors. This code creates two custom widgets through a custom widget class, TextJButton. Those two custom buttons are added onto a frame.

Since all the parameters for instantiating TextJButton class can be resolved by using a simple data-flow analysis, the jb1 and jb2 objects can be instantiated successfully. Finally, the two button objects are added onto the frame by using the hierarchy relationship statements in line 15 and line
16. These hierarchy relationship statements can be statically interpreted by the GUI editor. Using the “adding” information, the GUI editor is able to finish building the design model by adding two child nodes, jb1 and jb2, for the frame object. The final design model and the design view for the code from Swing Designer are presented in Figure 3.9.

![Diagram showing design view and widget hierarchy tree model](image)

(a) Design view                      (a) Widget Hierarchy Tree Model

Figure 3.9 [Example]: The design view (a) and widget hierarchy tree model (design model) (b) for the GUI code in Figure 3.8. The design view is rendered by using Swing Designer. In the design model, each node contains the object name and class name, which are separated by using a colon.

Table 3.2 [Example]: A list of types and methods from the code example in Figure 3.8, which have been hard-coded into the GUI editor.

<table>
<thead>
<tr>
<th>Types</th>
<th>JFrame, GridLayout, Dimension, String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
<td>main, setLayout, add,setText, setPreferredSize, setVisible</td>
</tr>
</tbody>
</table>

3.2.3  Defect 1 of Traditional GUI Editors

Now that we understand the code analysis algorithm that has been used by most traditional GUI editors, I will further explain defect 1 of GUI editors to motivate my research. In order to better explain this defect, I start by describing two kinds of programming techniques that are frequently
used in most GUI applications: GUI developers make heavy use of *dynamic GUI code* and *code abstraction* to create user interfaces for achieving simple and effective GUI programming. For each programming technique, I use some concrete code examples to demonstrate how this programming technique could create some serious problems for traditional GUI editors. Finally, the Java-Chess example is used to demonstrate why many widgets cannot be displayed, as shown in Figure 3.2.

### 3.2.3.1 Dynamic GUI Code

During the programming of GUI applications, GUI developers make heavy use of dynamic GUI code to program user interfaces. Dynamic GUI code is not static in nature, and the user interfaces that are generated by dynamic GUI code depend on the runtime calculations or runtime program inputs. According to my investigation of several existing GUI applications, a large portion of their user interfaces is implemented by using dynamic GUI code. There are mainly two kinds of dynamic GUI codes: the GUI code that contains control flow statements and the GUI code that depends on external sources or states.

The GUI code that contains control flow statements can change the execution flow of the GUI program through decision making, looping, and branching. Control flow statements include decision-making statements (*if*-then, *if*-then-else, *switch*), looping statements (*for*, *while*, *do-while*), and branching statements (*break*, *continue*, *return*) [96]. GUI developers often use control flow statements to generate user interfaces. For example, a chess board would require a developer to draw 64 squares. In this case, the developer is very likely to draw the 64 squares by using a loop instead of drawing them one by one.

The GUI code that depends on external sources provides a flexible way for developers to create different user interfaces based on different sources, states, or parameters. A good example is creating a user interface that is based on the parameters that are passed in the constructor.

The use of dynamic GUI code in the programming process is important and should certainly not be avoided for applications. However, this could create problems for traditional GUI editors. Existing GUI editors have problems dealing with dynamic GUI code because they have limited reverse-engineering capabilities.

When a visual class is opened by using a traditional GUI editor, the GUI editor tries to understand each line of the code. If the editor encounters any dynamic GUI code during the analysis of the visual class, it will skip the line of code and move to the next line. This analysis approach generates an incomplete design model because all the widgets (or properties) that are created or modified in the dynamic GUI code cannot be analyzed by the GUI editor. Furthermore, the design view is also incomplete because it is rendered based on the recovered design model. In this case, GUI developers
can only obtain very limited assistance by using this incomplete design view. In the next two parts, these problems are further demonstrated by using several concrete code examples.

**Example: GUI Code Containing Control Flow Statements**

In this part, a concrete code example from Java-Chess application is used to demonstrate the problems that are caused by control flow statements in the GUI source code. I chose a class called `ChessBoardRenderer2D`, which is used to generate the chess board as shown in Figure 3.10. However, when this class is opened by using a traditional GUI editor, the chess board cannot be visualized completely, as presented in Figure 3.5. The column labels and row labels are missing, as described in Section 3.1.

![Chess Board](image)

**Figure 3.10 [Example]:** The chess board generated by `ChessBoardRenderer2D` class. The columns (called files) are labeled by the letters a to h and the rows (called ranks) by the numbers 1 to 8.

This given class, `ChessBoardRenderer2D`, is a custom visual class. Therefore, the GUI editor starts by statically analyzing the only constructor of the given class. Now, let us look into the constructor code of the `ChessBoardRenderer2D` class to examine why the column labels and row labels cannot be displayed but the chess board can be visualized by the GUI editor.
1. public ChessBoardRenderer2D (... ) {
2. ...
3. //Add the row numbers to the board.
4. JPanel rowNumbers = new JPanel();
5. rowNumbers.setLayout(new GridLayout(8, 1));
6. for( int i = 8; i > 0; i--) {  
7.     rowNumbers.add(new JLabel("" + i, JLabel.CENTER));
8. }
9. add( rowNumbers, BorderLayout.WEST);
10. //Add the board itself.
11. JLayeredPane boardPane = new JLayeredPane();
12. boardPane.setPreferredSize(...);
13. boardPane.setOpaque(false);
14. //The squares of the board are drawn on the board.
15. boardPane.add(new BoardLayer(), JLayeredPane.DEFAULT_LAYER);  (b)
16. //Add the line names to the board
17. JPanel lineNames = new JPanel();
18. ...
19. byte [] name = new byte[1];
20. for( int i = 0; i < 8; i++) {
21.     name[0] = (byte)( 'a' + i);  (c)
22.     JLabel nameLabel = new JLabel(...);
23.     nameLabel.setPreferredSize(...);
24.     lineNames.add(nameLabel);
25. }
26. add( lineNames, BorderLayout.SOUTH);}

Figure 3.11 [Example]: The constructor code of ChessBoardRenderer2D class. The highlighted code snippet (a) generates the row numbers labeled 1 to 8. The highlighted code snippet (b) generates the chess board. The highlighted code snippet (c) generates the column numbers labeled a to h.
Figure 3.11 presents the code of the constructor from the ChessBoardRenderer2D class. When this class is opened by using a GUI editor, the editor starts to analyze this constructor’s implementation through the static analysis approach, described in Section 3.2.2. When the GUI editor encounters the code snippets (a) (Lines 6-8) and (c) (Lines 20-25) as shown in Figure 3.11, it cannot reverse-engineer them properly because the GUI editor cannot understand the loop statements that have been used to generate the labels for column panel and row panel. Therefore, the GUI editor just skips these statements and does not render any labels for the column and row panels of the chess board on the design view.

When the GUI editor analyzes the code in line 15, it encounters the allocation site that creates the chess board widget (new BoardLayer()). Since BoardLayer is a custom widget class, the GUI editor attempts to instantiate the BoardLayer class. The instantiation of this class does not require any parameters. Therefore, this chess board widget can be created successfully. Next, the instantiated chess board widget is added onto the boardPane in line 15, and this statement can be interpreted statically.

Through this code example, we can see that the traditional GUI editor cannot statically resolve control flow statements such as loops. In this case, the editor can only display an incomplete design view for GUI developers. In next part, I describe another kind of dynamic GUI code: the GUI code that depends on external sources.

**Examples: GUI Code Depending On External Sources**

Using static analysis to provide an abstract interpretation of the GUI is problematic. The reason is that the visualization of a widget object is highly sensitive to any missing details such as unknown constructor input parameters. The missing parameters sometimes carry very important GUI information for a GUI editor to render the widget object, which results in the widget not being rendered at all. In a limited number of cases, missing input can be handled reasonably, just as a traditional static analysis can abstract from missing input. For example, a GUI editor such as Swing Designer could choose to ignore a missing title parameter for a dialog window, but still show a meaningful visual representation for the rest of the window. However, if a missing input includes an entire set of widgets to be nested in the dialog, the displayed WYSIWYG view will have little semantic meaning for the developer.

Two code examples are provided to explain how the traditional GUI editor handles the missing parameter situations. In the first example (Figure 3.12), the code with a missing parameter can be dealt with by a traditional GUI editor, as shown in Figure 3.13; in the second scenario (Figure 3.14),
the GUI editor cannot handle the missing parameter situation, so it provides a design view without much semantic meaning for developers, as shown in Figure 3.15.

Suppose a GUI developer wants to use a GUI editor to make some changes to the GUI code that is presented in Figure 3.12. In the code, CustomizedFrame is a custom JFrame, and the title string of the frame is passed from another class testCustomizedFrame as indicated in line 19. This custom JFrame contains three buttons, and they are arranged on the frame through a special layout manager GridLayout.

```java
1. public class CustomizedFrame extends JFrame {
2.  public CustomizedFrame(String title) {
3.      this.setTitle(title);
4.      this.setLayout(new GridLayout(3,3));
5.      JButton button1 = new JButton("Button1");
6.      button1.setPreferredSize(new Dimension(100, 100));
7.      this.add(button1);
8.      JButton button2 = new JButton("Button2");
9.      button2.setPreferredSize(new Dimension(100, 100));
10.     this.add(button2);
11.     JButton button3 = new JButton("Button3");
12.     button3.setPreferredSize(new Dimension(100, 100));
13.     this.add(button3);
14.  }
15. }
16. }
17. public class testCustomizedFrame {
18.      public testCustomizedFrame(){
19.         CustomizedFrame cf = new CustomizedFrame("This is a customized frame");
20.      }
21. }
```

Figure 3.12 [Example]: The implementation for a custom JFrame. Three buttons are arranged on the custom frame (CustomizedFrame) with a layout manager, GridLayout. The title string of the frame is passed into its constructor from another class, testCustomizedFrame.

When the CustomizedFrame class as shown in Figure 3.12 is opened in a traditional GUI editor, the frame and buttons are rendered successfully by the GUI editor, as shown in Figure 3.13. By comparing the design view that is rendered by the GUI editor and the actual runtime appearance of the user interface, I find that the only difference is that the rendered design view does not have the
title “This is a customized frame”. The reason is that the title of this frame is passed from the class testCustomizedFrame. When CustomizedFrame is opened by using the GUI editor, the title string is missing. Therefore, the GUI editor produces an approximated design view of the actual appearance of the user interface. In this scenario, the approximated design view is quite close to the actual one, and the developer can still use this rendered design view to edit the frame, the three buttons, and the layout manager.

![Image of GUI editor](image.png)

**Figure 3.13 [Example]:** The rendered design view for the source code as shown in Figure 3.12 by a traditional GUI editor. Three buttons are arranged on the frame in a grid layout. The missing frame title is replaced with the tag `<dynamic>` because the GUI editor lacks the title string and cannot display it.

In another scenario, the developer is asked to change a different implementation of the CustomizedFrame class, which is presented in Figure 3.14. In this new implementation, the title string of the custom frame is fixed with a string “This is a customized frame” as indicated in line 4. The three buttons arranged on the frame are based on the layout manager, which is passed to the constructor at line 5.

The two implementations for the CustomizedFrame class as shown in Figure 3.12 and Figure 3.14 have the same user interface appearance at runtime. However, when the CustomizedFrame class in Figure 3.14 is opened through the same GUI editor, the three buttons cannot be visualized by the GUI editor, as shown in Figure 3.15.
The implementation for a custom frame. Three buttons are arranged on the custom frame by using the layout manager that is passed into the constructor of this custom frame class (CustomizedFrame). The frame title is fixed in line 4.

Figure 3.15 [Example]: The rendered design view for the source code as shown in Figure 3.14 by the same traditional GUI editor. Three buttons are missing from the frame because the GUI editor lacks the layout information.
For the code in Figure 3.14, the traditional GUI editor lacks the necessary parameter to understand how to arrange the three buttons on the frame. In this scenario, the three buttons cannot be rendered by the GUI editor at all. Traditional GUI editors could use different strategies to handle this situation; for example, some GUI editors still render all the buttons that are added on the frame but with a default layout manager (e.g., BorderLayout). However, some GUI editors choose to be more conservative by not displaying any buttons on the frame. In this case, GUI editors have a problem understanding the GUI code that depends on external sources, states, and parameters.

3.2.3.2 Code Abstraction

Based on my investigation of several open-source GUI applications, I have seen the following major distinction between small GUI applications and larger GUI applications, in respect to the way in which the GUI source is written. For small GUI applications, the mapping between the source statements and the runtime widgets is easy to identify. In most cases, it is possible to make a one-to-one mapping between statements in the source and the widgets at runtime. For large applications, the mapping is very often one-to-many, e.g., one statement could be mapped to many runtime widgets. The reason is that GUI developers of large applications make heavy use of programming abstractions to improve the scalability of the development process.

For example, at runtime, the typical GUI part of a program will consist of a finite set of windows, each of which has a unique identity from the perspective of the user. In an application, common examples of window identities are: “Main window”, “About window”, “Options window”, and “Open file window”. Although the details vary, the important point is that these windows are implemented in a similar way. Based on my observations of several open-source GUI applications, developers can often reuse code among different windows. As an example, I investigated the GanttProject application [38], an open-source Java application for managing Gantt charts. In GanttProject, there is a TopPanel class. Several dialog windows from GanttProject make use of this class to provide a standardized look across the dialogs in the application.

In the GanttProject application, each window consists of a (possibly) nested set of widget containers and components. These widgets are distinguished easily from each other at runtime because they appear in the context of distinct windows. However, it may be difficult for developers to distinguish between different widgets by examining the source code. For example, a set of widgets could share similar appearance and behavior. These widgets could be created based on different subclasses that are inherited from the same superclass. In this case, some source code in the superclass could be shared by all the subclasses.
Therefore, in large-scale applications it would not be practical for every GUI element observed during program execution (e.g., window, button) to map one-to-one with a syntactic program element (e.g., statement). In order for software development and maintenance processes to be scalable, developers must use abstraction features such as parameterization and inheritance during development. As program sizes increase, the amount of programming language abstraction applied to the construction of the GUI increases.

Based on my observations, I have identified two common sources where code abstractions are heavily used: function-based abstraction and class-based abstraction.

First, the function-based abstraction is used to decompose a program into some smaller and more manageable software modules. This software decomposition is useful for reusing the software modules in different places of the program. Furthermore, parameterization could be used to further allow developers to reuse the software modules in different contexts. For example, in some GUI applications, the creation of some widgets is encapsulated in a helper function. Inside the function, some program logic is implemented to create different widgets that could have slightly different appearances, and this difference is controlled by the parameters that are passed into this function.

Second, as described earlier in this sub-subsection, developers of large applications often create custom widget classes, and these classes directly extend from classes exported by the GUI frameworks. Furthermore, as applications become larger, it is important to abstract common details between widgets using inheritance. Therefore, a common part of the visual appearance for each subclass is refactored into a common superclass. For example, in an open-source GUI application JEdit, there are 16 classes whose names end with OptionPane that all inherit from the class AbstractOptionPane. This superclass defines some visual appearance that is common to all subclasses. Each subclass adds additional visual appearance, which is controlled by passing different parameters into the constructors of the subclasses. I call this kind of code abstraction class-based abstraction.

The code abstraction mechanisms, such as function-based code abstraction and class-based code abstraction, are definitely useful and should not be avoided by developers. To accommodate this need, most editors function by generating source code from the GUI design, which can be tailored manually to introduce abstraction. In this way, the visually designed prototypes of widgets can be made into reusable source code components. For example, a specialized dialog class might provide a “template” for creating dialog windows with some common properties (e.g., size, layout, look, feel), but take some parameters as input for other properties (e.g., title, the widgets to be nested in the dialog). Unfortunately, the code abstractions have negative side-effects: once code abstractions are
applied to the generated code, a GUI editor will have difficulty being able to visualize any part of the semantic design. Next, I explain this difficulty by using a concrete code example.

**Example: Code Abstractions**

Traditional GUI editors provide limited support for code abstractions. For example, as explained by Swing Designer in [92]: “[The code that has] multiple references to the same widget definition through multiple invocations of the same helper method [cannot be rendered].”

```
1. public class DemoColorButton {
2.     public static void main(String[] args) {
3.         JFrame frame = new JFrame("A text button sample");
4.         //Create two button texts
5.         String button1 = "button1";
6.         String button2 = "button2";
7.         //Create a JButton with the button text "button1"
8.         final JButton jb1 = createTextButton(button1);
9.         frame.add(jb1, BorderLayout.WEST);
10.        //Create a JButton with the button text "button2"
11.        final JButton jb2 = createTextButton(button2);
12.        frame.add(jb2, BorderLayout.EAST);
13.        frame.setSize(300, 200);
14.        frame.setVisible(true);
15.    }
16.    private static JButton createTextButton(String name) {
17.        JButton button = new JButton();
18.        button.setText(name);
19.        return button;
20.    }
21. }
```

**Figure 3.16 [Example]:** The code example demonstrates the usage of function-based code abstraction. The two text buttons are created by using a helper method, `createTextButton`.

The theoretical reason behind this restriction is that traditional GUI editors can only reverse-engineer widget instantiating statements that have one-to-one mapping relationships with runtime widgets, i.e., one statement only maps to one widget. However, the use of code abstractions often
results in one-to-many mapping, i.e., one line of GUI code generates more than one widget. In this case, if GUI editors allow these widgets to be rendered for developers, it could create a serious problem for the developers to edit these widgets. The reason is that editing one widget could cause changes applied to all other widgets that share the same code. Sometimes this change is not desirable. This problem is further explained in Section 3.4, when view-based editing is discussed.

For example, Figure 3.16 presents a code example that uses function-based code abstraction. This example generates a frame that contains two buttons. The two buttons have different text, “button1” and “button2”. In this GUI code, a frame widget with the title “A text button sample” is first created in line 3. Next, two custom buttons are created by the same allocation site in line 17 and then added onto the frame widget in line 9 and line 12. Here, the code is written by using function-based code abstraction: the creation of different custom buttons is refactored into a helper method called createTextButton. In this code example, the allocation site in line 17 maps to two button widgets at runtime. Therefore, a traditional GUI editor such as Swing Designer cannot properly render this code because of this one-to-many mapping. The generated design view is incomplete, as shown in Figure 3.17. The two buttons are missing in the design view because the GUI editor has limited capability to handle function-based code abstraction.

From the above example, we can see that GUI editors provide limited support for function-based abstraction. This is also the case for class-based abstraction, and its code example would be very similar to the example that has been given in Figure 3.16. In this case, the code example will not be provided, to avoid repetition.

![Figure 3.17](image_url)

**Figure 3.17 [Example]:** The design view for the code in Figure 3.16. The frame is rendered, but the two buttons are missing because the traditional GUI editor has limited support for function-based code abstraction.
3.2.3.3 The Java-Chess Example

Having presented the limitations of the analysis capabilities of traditional GUI editors, I will use the user interface example from Java-Chess (Figure 3.1) to further demonstrate why so many widgets cannot be rendered as shown in Figure 3.2. Figure 3.18 presents a complete widget hierarchy tree for the Java-Chess user interface. In the tree, all the nodes represent the widgets that are on the user interface. The solid lines represent the relationships that can be recovered by traditional GUI editors. I call these relationships *static relationships*. The dotted lines represent the relationships that can only be recovered at runtime. I refer to these relationships as *dynamic relationships*. For example, the 8 column labels are assembled and put on the column panel through a loop, which cannot be reverse-engineered by using a static analysis approach. Therefore, the dynamic relationships between the column panel and the 8 column labels are marked with dotted lines in the widget hierarchy tree, as shown in Figure 3.18.

![Figure 3.18](image)

*Figure 3.18 [Example]:* The widget hierarchy tree for the Java-Chess user interface. All the nodes represent the widgets in the Java-Chess user interface. The solid lines represent static relationships that can be recovered by traditional GUI editors. The dotted lines represent dynamic relationships that can only be recovered at runtime.

Traditional GUI editors cannot properly analyze the dynamic relationships. Therefore, they are blocked in the dynamic parts. In the end, the entire hierarchical view becomes truncated. Sub-trees are no longer visible in context because they are not reachable from the root widget in a design view. Figure 3.19 presents the truncated widget hierarchy tree. In this new tree model, all the widgets that
are directly connected to the tree model through dynamic relationships and all the corresponding child nodes (not reachable from the root widget) of the widgets are deleted from the original tree model.

Comparing Figure 3.19 with Figure 3.18, we can see that a large number of widgets are removed from the widget hierarchy tree in Figure 3.18, and all the removed widgets represent the widgets that cannot be displayed by traditional GUI editors. Therefore, traditional GUI editors have been placed onto a really cumbersome position. It is very helpful to use traditional GUI editors to assist GUI developers in understanding a large amount of GUI code that is scattered across many source modules. However, the precondition is that GUI editors have to be able to work properly. Based on the Java-Chess example and several other examples I have investigated, traditional GUI editors have poor capabilities in reverse-engineering, which make them unable to provide much assistance in solving the two problems for GUI understanding and maintenance. In the next two sections, I present my solutions to this problem.

![Widget Hierarchy Tree](image)

Figure 3.19 [Example]: The truncated widget hierarchy tree. Compared to the tree in Figure 3.18, all the widgets directly connected to the tree model through dynamic relationships and all the corresponding child nodes of the widgets (not reachable from the root widget) are deleted. This truncated widget hierarchy tree represents all the widgets that can be displayed by traditional GUI editors.

### 3.3 View-Based Navigation

View-based navigation provides hyper-linking between the GUI editor and source code locations. First I explain this relatively straightforward “read-only” support of my tool. Then in Section 3.4, I build on these details to explain the more complicated “read/write” editing support of my tool.
As shown in Figure 3.20, my approach makes use of information derived from the dynamic analysis of program execution by using AspectJ. My Eclipse IDE plug-in called FreezeFrame [55][56] creates a dynamic model of the GUI, effectively replacing the design model normally used by the editor (as in Figure 2.1, (b)). A dynamic model is created by capturing the state of widgets at a particular time. This state of a widget consists of widget properties and possibly other child widgets.

For my tool to create a dynamic model of some GUI, a developer is required to execute the program, making sure the dynamic view the developer is interested in is rendered during that execution. Next, my tool uses a dynamic analysis approach to build a dynamic model. After this analysis, the developer can navigate between widgets in the dynamic view and source code elements using the plug-in. This provides a complete, dynamic context for view-based navigation.

![Figure 3.20: During the execution of a GUI application, a developer chooses a specific user interface that they are interested in. Then the dynamic model of the user interface is captured by using the advice of a monitoring aspect. Next, the static design model that is normally used by a GUI editor is replaced with this dynamic model.](image)

During program execution, interception of method call join points specific to the GUI concern is used to create a dynamic model. I use AspectJ to collect this information about the GUI. I created an aspect, GUIAspect, to monitor three important kinds of events related to the user interface. For my
prototype, applications using the Java Swing API [90] are supported. In Table 3.3, I provide simplified pointcuts from my implementation to illustrate.

In each advice, the reflective capabilities of AspectJ are used to record information for argument values and corresponding source locations (i.e., classes and line numbers). This information is used to create a map between widgets and source code, to support view-based navigation.

Once the developer is satisfied that the view they are interested in has been displayed during execution, the user can select an option from my plug-in, which will in turn provide the currently constructed dynamic model to a GUI editor. The actual underlying data in the design model is now replaced with a dynamic model. I used this approach simply to avoid making a custom GUI editor from scratch. Now when the user selects widgets in the GUI design view, they can choose to be hyper-linked to the source code statement of the corresponding GUI join point as captured by GUIAspect.

Table 3.3: Simplified pointcuts for FreezeFrame and corresponding explanation.

<table>
<thead>
<tr>
<th>Pointcuts</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>call(javax.swing.*+.new(..))</td>
<td>The creation of widget classes is intercepted according to this construction pointcut.</td>
</tr>
<tr>
<td>call(void javax.swing.<em>.set</em>(..))</td>
<td>Information regarding any changes to the properties of a widget is captured through the pointcut.</td>
</tr>
<tr>
<td>call(* javax.swing.*.add(..))</td>
<td>Information regarding the containment relationship between container widgets and their child widgets is captured through this pointcut.</td>
</tr>
</tbody>
</table>

In my evaluation (Section 3.6), I quantify the number of classes that contribute to the generation of a typical dynamic view for open-source applications. Since source locations across all these classes are now made available to a developer from a single high-level conceptual module (the view), this metric helps provide some measure of the reduction in information that must be reasoned about by the developer for navigation.
3.3.1 Custom Graphics

In many user interfaces, part of the dynamic view may consist of custom animations or graphics generated dynamically using calls to a graphics package. These graphics do not consist of widgets but rather are translated directly to a two-dimensional array of pixel elements to be displayed on screen. In Java, custom graphics are implemented using objects that support a “callback”:

\[
\text{public void paint(Graphics g);}\]

The implementation of the method makes use of the graphics object to draw custom raster graphics. GUIAspect advises this method using an after advice and captures the graphics argument. After paint has executed, the body of the method will have written information to the object. I extract this information and save it on disk as a graphics file. Then I place an Image widget in the model as a “proxy” for the custom graphics object, so this image is displayed when the dynamic model is rendered by the editor.

During view-based editing (Section 3.4), this provides the developer with a “snapshot” of the animation or graphic. This is useful for providing a dynamic context to inform maintenance decisions. For example, in the Java-Chess application (Figure 3.1) the entire chessboard is drawn using graphical rectangles. The chessboard squares are not widgets, but the chess pieces are widgets. So the chessboard cannot be edited from a design view but the chess pieces could. However, without the dynamic context of the chessboard, it would be difficult for a developer to make an informed design decision about potential changes to the chess pieces. This shows how even “read-only” information can be useful to provide a dynamic context for view-based editing, which I describe next.

3.4 View-Based Editing

When developers edit widgets on a design view, the design view and the source become inconsistent. To perform round-trip engineering, I have to find an inconsistency handling policy to reconcile the design and source. First, I discuss the technical difficulties of GUI round-trip editing in the context of a concrete code example to explain why the reconciliation is difficult. Then I present my policy as implemented.

Figure 3.21 shows the Java-Chess source code for generating the chessboard column labels seen on the bottom of the chess board in Figure 3.1. These labels indicate the columns a to h. This
example demonstrates a situation in which a one-to-one correspondence between source and a design model cannot be created. In this case I must warn against editing from the design view.

In the source code, we see that column labels are generated by a loop (lines 2-7). The characters to be displayed are created using the loop counter and adding it to the program constant ‘a’ using arithmetic addition of character codes (line 3). Then each label widget is created and its text property is set (lines 4-5). Finally, each label is added to the panel (i.e., this) on line 6.

```
1. ...
2. for(int i = 0; i < 8; i++) {
3.   char column = 'a' + i;
4.   JLabel label = new JLabel();
5.   label.setText(column+"" );
6.   this.add(label);
7. }
```

Figure 3.21 [Example]: Java-Chess source code in the constructor of chess board column panel.

Code has been refactored to improve readability.

![Chess board column panel](image)

```
1. ...
2. for(int i = 0; i < 8; i++) {
3.   char column = 'a' + i;
4.   JLabel label = new JLabel();
5.   label.setText("G");
6.   this.add(label);
7. }
```

Figure 3.22 [Example]: A developer changed the label ‘g’ to the label ‘G’ in the design view using FreezeFrame, if it naively translated line 5 to `setText("G")`, we would end up with eight identical labels.
It is very difficult (or even impossible) to translate a change in one of the characters a to h from an editor design view to some underlying change on the source, without completely disrupting the source structure.

Suppose a developer changed the label ‘g’ to the label ‘G’ in the editor design view with my dynamic model installed. If the editor naively translated line 5 to `setText("G")` as shown in Figure 3.22, we would end up with eight identical labels!

Recall that a design model is simply a tree data-structure of widgets. Considering line 5, an editor could not create a one-to-one correspondence between this statement and a design model for two reasons. First, the `label` variable refers to more than one object during the constructor execution. Second, the `column` variable does not refer to a GUI constant. Even though my tool can display such computed information, I have to respect the limitations of graphical-based editing.

This one-to-many correspondence between one statement and a design model can be quite common in GUI applications; for example, consider DrawSWF [27], a drawing/animation program. The user interface includes a toolbar with a number of icons. The toolbar acts as a palette from which the user can select different drawing tools (e.g., pencil, text, rectangle tool). The tool icons are positioned in such a way that related tools are adjacent. These widgets were actually created as a few different clusters, so that changing the properties of one tool icon in a cluster will affect the other tool icons in that cluster. By examining the source code, I verified that in fact three different `for` loops were used to generate the tool icons, so that each loop generated a different cluster.

In many of the cases where I have seen this type of clustering, there appeared to be a very obvious semantic relationship among all the members in the cluster, which were induced by the way the source code was written. Currently, there is no existing GUI editor that can help point out these semantic relationships in the design view. Source code patterns (such as creating a set of items in a loop), cannot always be used to infer meaningful semantic relationships. Although my dynamic GUI editor always displays the relationships through a design view to the developer, it is up to the developer and not my tool to judge whether the relationship is meaningful. They need to explicitly check the source codes. If a change is desired for the entire set of instances, the developer can now apply changes uniformly across all members of the set. If a developer wishes to single out a particular widget for editing, they should apply source code changes to affect the loop input and not directly to the loop body. In this case, my tool restricts developers from using WYSIWYG editing, and they have to check the actual source code to decide what code changing is semantically meaningful. Therefore, my research has to provide an editing constraint to restrict view-based editing, which is discussed in the next subsection.
3.4.1 Editing Constraint

When any changes are made on a design view, traditional GUI editors reconcile the design model and the source. In traditional GUI editors, only changes on the design view which correspond one-to-one with source code changes can be performed. The reason is that only functions that are a one-to-one correspondence have a well-specified inverse [19][60]. However, since my tool has allowed the developer to view dynamic context, my tool has to provide guidance to prevent accidental editing of dynamic information. Traditional GUI editors also must wrestle with this fundamental problem. Their approach is to not display any dynamic information, as discussed in Section 3.2.2. They only display the information for which a one-to-one correspondence has been created. In the remaining part of this subsection, I first define a concept that has been used by most traditional GUI editors, change context, and then I provide my editing constraint.

3.4.1.1 Editing Change Context

When a change is made on a widget, a traditional GUI editor translates the change on the widget to the source location immediately before the statement where this widget is added on the design model. The visual class where the statement is located provides the scope for changes to the widget. I call such a class change context.

3.4.1.2 Constraint Details

I have investigated the use of an editing constraint. For this editing constraint, my implementation is an adaptation of the existing static design model recovery technique used by GUI editors (Section 3.2.2).

When a developer makes a change to the design view with my plug-in activated, that change is translated by the editor into a change to a widget. This widget will be part of the dynamic model collected by GUIAspect. A source code edit will be made by the editor to realize the change, if it is not prevented by my constraint.

Since change context is the scope for a source code change on a particular widget, my tool first determines the potential change context for the change. The change context is easy to obtain because the dynamic model that my tool captured at runtime records the source location (class name and line number) where the widget is added on the dynamic model, as described in Table 3.3.

Next, the identified change context is analyzed by using the traditional GUI editor’s static recovery technique, as described in Section 3.2.2. If the change context class only has one constructor, the static design model is recovered (line 9 of Figure 3.23) by using the algorithm as
described in Figure 3.7. If the change context class contains multiple constructors, my implementation uses the union of static models recovered from all class constructors (lines 4-7 of Figure 3.23). This is useful because I noticed that often multiple constructors are declared, which simply delegate construction of the GUI to a single private helper method.

Finally, my tool can simply compare the sub-tree of the dynamic model rooted at the object of the change context class (line 11 of Figure 3.23) and the recovered static design model of the change context. My tool has to determine if they “intersect” at the widget where editing is considered. If they do agree, then the change is allowed; otherwise, the change is prevented and a warning is issued (line 13 of Figure 3.23). The reason is that a static design model contains all the widgets that have one-to-one mapping with the source statements. All these widgets can be edited safely. Therefore, if a design model and a dynamic model intersect at a widget where the editing is considered, then this widget can be edited safely.

```
1. Input: widget, changeContext, dynamicModel
2. BEGIN
3. IF |changeContext constructors| > 1 THEN
4. FOR each constructor of changeContext constructors
5. constructorStaticModel = constructor.StaticModel
6. staticModel = staticModel \cup constructorStaticModel
7. END FOR
8. ELSE
9. staticModel = changeContext.constructor.StaticModel
10. END IF
11. subDynamicModel := dynamicModel.getSubtree(changeContext)
12. intersectedTree := TreeToTreeComparison (staticModel, subDynamicModel)
13. isEditable := intersectedTree.isContain(widget)
14. RETURN isEditable
15. END
16. Output: isEditable
```

Figure 3.23: This editing constraint used by my tool. This constraint is used to determine if a widget on a design view can be edited safely.
This intersection is implemented with top-down tree-to-tree comparison [81] (line 12 of Figure 3.23). In the future I could provide a visual cue such as a highlighting or overlay that points out all of the static design information embedded in some dynamic view.

Next, I use a concrete code example to demonstrate this editing constraint and explain how this editing constraint is used to restrict a developer’s ability to alter some dynamic parts of the design view. I use the chess board example and its implementation, ChessBoardRenderer2D class, to illustrate this editing constraint.

After executing the Java-Chess application, a developer sees a complete design view for the main window of the Java-Chess program by using the dynamic GUI editor. In this design view, they can see the chess board completely. If they want to edit the label ‘a’ on the column panel of the chess board, the editing constraint should be able to prevent any modifications to this widget for the reasons discussed. Next, I use the label ‘a’ as an example to illustrate the editing constraint.

When the developer tries to edit the widget label ‘a’, my editing constraint will first have to figure out the potential change context for this change. As explained, my tool determines the change context by finding the source location where the label ‘a’ is added on the dynamic model. Recalling Figure 3.11, which presents the constructor code for the creation of the chess board, we can see that the located change context for the label ‘a’ is ChessBoardRenderer2D.

![Diagram](image)

**Figure 3.24 [Example]:** The static design model of ChessBoardRenderer2D class, which has been recovered by Swing Designer. In each node of the tree, the name of the widget and the corresponding type information are presented and separated by a colon.
The next step of the editing constraint is to analyze the constructor of the located change context through the static recovery technique that has been used by most traditional GUI editors. After this analysis, the static design model of the change context is obtained.

This static design model for a specific visual class is easy to obtain from traditional GUI editors because they often disclose this information to GUI developers to help them in understanding the widget hierarchy relationships; for example, the left side of Figure 3.24 provides an example of the static design model for the ChessBoardRenderer2D class, which has been recovered from Swing Designer. On the right side of Figure 3.24, this static design model is presented as a tree.

![Diagram of static design model](image)

**Figure 3.25 [Example]:** A sub-tree of the dynamic model for the main window of the Java-Chess application (rooted at the chess_Board widget). In each node, the name of the widget and the corresponding type information are presented and separated by a colon.

Finally, the editing constraint can simply compare the sub-tree of the dynamic model rooted at chess_Board (the object of the change context class) as shown in Figure 3.25 and the static design model of the change context class as shown in Figure 3.24. Clearly, the design model in Figure 3.24 does not have an intersection with the sub-tree of the dynamic model in Figure 3.25 at the widget label ‘a’. So the change is denied and a warning message is presented to the developer.
The editing constraint is important to prevent developers from editing the dynamic design of user interfaces. In the next subsection, I present how my dynamic editor that uses this editing constraint could improve view-based editing compared to traditional GUI editors.

3.4.2 Dynamic Context Improves View-Based Editing

Using my tool, a developer sees a dynamic context instead of a static one. This dynamic context contains both static design information and dynamic information. Editing the static design information should be allowed. Editing the dynamic information from the editor requires constraints to aid developers, although view-based navigation still works as originally described. I showed in the previous subsection why some restrictions on editing dynamic information make sense.

![Diagram](image)

Figure 3.26 [Example]: High-level illustration of a dynamic model (object graph) including design (solid) and dynamic (dotted line) information. Numeric labels refer to “add” relationships between widgets in Figure 3.27. A design model will appear to the user as a truncated version of a dynamic model.

Editing static design information is also supported by traditional GUI editors. So it may seem at first glance that the only thing I have contributed is view-based navigation for dynamic information. This is not the case. Here, I explain how my tool can help provide more context for editing design information.

To make this clear I provide a simple abstract example as shown in Figure 3.26. Here, there are four widgets: frame, panel, label, and button. The lines that are solid are static relationships derivable from a design model. The lines that are dotted are dynamic relationships, derivable only from a dynamic model. The source code creating these relationships is shown in Figure 3.27.
Figure 3.27 [Example]: Three custom widget classes used to demonstrate how dynamic context improves view-editing (an illustration is shown in Figure 3.26). Notice that, since the argument in the MyFrame constructor requires knowledge of the dynamic type, it cannot be displayed in a traditional design view.
This scenario corresponds to the source code in Figure 3.27. When the frame is created in the main method, a MyPanel is passed into the frame’s constructor as an actual argument. However, since the JPanel formal argument of the MyFrame constructor could refer to any custom subclass of JPanel, a traditional editor would only display a single frame and a label (no panel would be displayed). Programs include many sources of information that can only be determined at runtime. Here I just use the dynamic type of an object as one example.

Using a traditional GUI editor, the frame could be viewed for editing because it is a custom widget, although no panel or button design would ever be shown as contained inside the frame. A developer interested in maintaining a particular view of the user interface may find it frustrating not to be able to edit seemingly static design portions of MyPanel in the visual context of the frame, for example, the button.

This demonstrates an important point from a software maintenance perspective. Hierarchal structures, such as the widget hierarchy structures, provide a disciplined way to organize modules so they are easier to navigate. However, as I showed, the hierarchy displayed in existing editors can become truncated, leaving the user with a “flat” set of modules through which no relationships are provided.

A developer using my tool could execute the program under their control until some dynamic view was rendered that they needed to maintain. They might even control the program in such a way as to create relationships that would be helpful for them during the maintenance process. For example, if more than one kind of panel could be placed in the frame of Figure 3.26, they could choose to place the one they were interested in. They could perform view-editing or view-navigation in this visual context of their choosing. Certain dynamic information could not be edited from the design view. I provide some initial quantification of these potential advantages for specific applications and use-cases, in Section 3.6.

### 3.5 Dynamic GUI Editor

In this section, I walk through three steps to see how the developer uses my dynamic GUI editor (FreezeFrame) to help her in completing the three tasks, described in Section 3.1.

1. Run Aspect Weaver for Java-Chess
   This step weaves aspects into the Java-Chess program to monitor the three important kinds of events related to the user interface, as described in Section 3.3.
Figure 3.28 [Example]: A complete design view of the Java-Chess main window. All the missing widgets are recovered by using my analysis. Sub-trees lost in Figure 3.2 have been “glued” back on.

2. Execute Program
Next, the developer executes the Java-Chess program to collect information about the GUI. Once the important widgets for her tasks are rendered, she notifies the plug-in to make a “snapshot” of the current dynamic model to replace the static design model used by the editor. In this case, when the developer runs the Java-Chess application, she will see the engine menu, the navigation panel, and the chess board labels when the Java-Chess main window is rendered as shown in Figure 3.1. After that, she informs the plug-in to make a snapshot.

3. Edit on the Complete Design View
When the developer opens the editor, she will be able to see a complete view of the main window as shown in Figure 3.28. The developer will be able to add the key code for the “3 seconds” option, which is under the Engine menu, since she easily finds the editable menu item with the help of my analysis. Furthermore, the developer can edit the navigation panel in the complete context (the Java-Chess main window) instead of trying to make changes in an isolated view of the panel. Now the
developer will easily notice the layout constraint for the navigation panel. So, she changes the layout constraint and then puts the hint button in the desired position.

Figure 3.29 [Example]: A code comparison view for the NavigationPanel class is provided to the developer. The difference between the two versions of the code is highlighted. The developer can confirm this change by clicking on OK or discard this change by clicking on Cancel if she finds any mistake.

During the code changing, a code comparison view is provided as shown in Figure 3.29. The developer can compare the refactored source with the original source for the visual class of the navigation panel. Next, she can either confirm this code modification by clicking on OK or discard this modification if she finds any mistakes of this code modification. I provide this extra code comparison view because my tool now allows developers to work on any handwritten GUI code instead of only well-structured GUI code that has been generated by using traditional GUI editors. In this case, a code comparison view is very helpful to provide a chance for GUI developers to check if this code modification will create any problems for the existing handwritten GUI code.

Finally, when the developer wants to edit the column labels, she will find that they cannot be edited, as discussed in Section 3.4. My static checking warns her about modifying these dynamically
computed properties. A window will be presented, as shown in Figure 3.30, to notify her that this widget modification is restricted because the widget is generated in a dynamic way.

![Image of Java Chess applet](image)

**Figure 3.30 [Example]:** The popup notification is presented when a developer tries to modify any widgets that are dynamically generated.

```java
for (int i = 0; i < 8; i++) {
    byte = (byte) \(('a' + i)\);
    JLabel nameLabel = new JLabel(new String\[\] name), JLabel.CENTER));
    nameLabel.setPreferredSize(new Dimension\[\] squareSize, squareSize / 2));
    lineNames.add(nameLabel);
}
```

**Figure 3.31 [Example]:** Design/code hyper-linking brings the developer to the statement where the column labels are generated.

However, the enabled design/code hyper-linking still works. When the developer clicks on any labels on the column, she will be brought to the statements that generated the labels, as shown in Figure 3.31. Next, she can modify the code manually.
By using my approach, the developer could maintain the user interface in a complete and context-aware design view. I discussed how my approach helped the developer with her three tasks where the traditional approach had failed. The modified result that satisfies all three user requests can be seen in Figure 3.32.

![Figure 3.32 Example: All the user-proposed requests are satisfied by using the dynamic editor. A key code “F1” is added for the “3 seconds” option, which is under the “Engine” menu. A new “Hint” button is added onto the navigation panel, and the labels of the columns are changed manually to support the descriptive notation.](image)

### 3.6 Quantitative Evaluation

I provide a quantitative evaluation of my approach using five open source applications. First, for some specific program views, I want to measure how many classes contribute implementation for generating the view. This metric could support my claim that maintaining GUIs can be difficult and that view-based navigation is useful. Second, I want to measure how much of some view for each
program could be reverse-engineered using a GUI editor. This could support my claim that design code and program logic is often tangled, making design model recovery difficult. Third, I wanted to compare the amount of static design information available in a particular use-case, using the original and my proposed approach. This could support my claim that dynamic context provides more opportunities for design view-editing.

Table 3.4: Application subjects and their sizes in terms of non-commented lines of code

<table>
<thead>
<tr>
<th>Application Name</th>
<th>NCLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrosswordSage</td>
<td>3,093</td>
</tr>
<tr>
<td>Java-Chess</td>
<td>5,616</td>
</tr>
<tr>
<td>jMSN</td>
<td>7,335</td>
</tr>
<tr>
<td>GanttProject</td>
<td>43,338</td>
</tr>
<tr>
<td>FreeMind</td>
<td>65,420</td>
</tr>
</tbody>
</table>

I initially looked at three existing Eclipse-based GUI editors for comparison with my approach: Swing Designer [91], Visual Editor [45], and Jigloo [15]. I chose Swing Designer to extend and compare against my approach. I discovered from their documentation that all the editors use the same overall approach for static model recovery, described in Section 3.2.2.

Swing Designer and Jigloo performed consistently better than Visual Editor. They are both mature commercial products whereas Visual Editor is an emerging open source project. Swing Designer out-performed Jigloo so I only present metrics from Swing Designer. I have also found an online article that rates Swing Designer very highly [86] against other editors. These reasons make me confident that the tool I am comparing against is a fair representative of existing tools.

For the evaluation subjects, I selected five open source applications which are built on the standard Java GUI libraries. My first subject, Java-Chess, was selected before my prototype was developed and was used to inform the creation of my approach. Four other subjects (CrosswordSage[18], jMSN[47], GanttProject[38] and FreeMind[35]) were selected after my prototype was developed. These programs were selected because they were used as subjects in a paper about testing GUI programs [106]. By using subjects selected by a third-party I hoped to provide objectivity in the
results. Table 3.4 lists the names (column 1) and sizes (column 2) of each subject in terms of non-commented lines of code (NCLOC).

Since my approach is based on dynamic analysis I needed to choose some specific moment in the execution of each program for which to take measurements. I wanted this choice to be as unbiased as possible and to be as uniform as possible across all the programs. I noticed it was very common for Java applications with a GUI to include a default “main window” with a variety of panels, menubar, and toolbars. This main window and the widgets it contains are often loaded immediately upon execution of the program. Then, after the main window is rendered, the application becomes idle, waiting for user input. I felt this point in time where the application becomes idle made a good choice of time because it was easy to objectively determine this point in the execution of many different programs. I call the execution of a program up to this point, the “main window scenario”.

3.6.1 Decomposition of Main Window View

Table 3.5: This metric measures the number of classes containing GUI join points captured by GUIAspect for rendering of the main window view

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Classes Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrosswordSage</td>
<td>2</td>
</tr>
<tr>
<td>Java-Chess</td>
<td>16</td>
</tr>
<tr>
<td>jMSN</td>
<td>11</td>
</tr>
<tr>
<td>GanttProject</td>
<td>45</td>
</tr>
<tr>
<td>FreeMind</td>
<td>12</td>
</tr>
</tbody>
</table>

The metric in Table 3.5 measures the number of classes containing method call join points captured by GUIAspect for rendering the window of the main window scenario. This provides me with an indicator of the reduction of source classes that a developer must manually navigate when performing maintenance on this particular view of the program. Since classes are the primary unit of modularity in object-oriented design, a larger number of classes indicates more difficulty performing maintenance on the view without proper tool support. These results are displayed in Table 3.5. From
the results, we can see that in at least 4 out of the 5 cases (where over 10 classes were used) there is evidence that view-based navigation would be useful.

### 3.6.2 View-Navigation

For the metric in Table 3.6, I measure the “size” of the dynamic model collected by GUIAspect and the size of the static design model recovered by the original GUI editor, for the main window view. This size metric was determined by the number of widgets and number of set widget properties (e.g. `button.setText("open")`) used to render the view. In each case, it is the size of the widget hierarchy tree model in terms of these logical elements.

**Table 3.6: Size metric is determined by the number of widgets, and set widget properties used in the dynamic model and the static design model**

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Size of dynamic main window model</th>
<th>Size of original design model relative to dynamic model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrosswordSage</td>
<td>143</td>
<td>100%</td>
</tr>
<tr>
<td>Java-Chess</td>
<td>291</td>
<td>45%</td>
</tr>
<tr>
<td>jMSN</td>
<td>149</td>
<td>34%</td>
</tr>
<tr>
<td>GanttProject</td>
<td>491</td>
<td>8%</td>
</tr>
<tr>
<td>FreeMind</td>
<td>109</td>
<td>10%</td>
</tr>
</tbody>
</table>

The actual dynamic model was determined by using GUIAspect and counting the size of the actual main window component graph through a traversal of objects. The models can seem quite large upon observation. I note that in addition to the information seen directly in the main window, the models include information about all menu items (and sub-menu items) in all drop-down menus of the window. Also, GanttProject is almost twice the size of the second largest model because the main window contains two “tabbed panes” with different user interfaces.

To record results for the original GUI editor I needed to know the custom class which represented the top window for the main window scenario. This was easy to determine for all cases and was usually the class containing the `main` function. I input this class to the original editor and recorded...
the numbers. These numbers are available from the tool itself. It is useful to note that all of the information displayed by standard editors is statically editable because they only display information recovered using static reasoning.

These results are displayed in Table 3.6. For one application, CrosswordSage, the complete main window view was recovered statically. This is because CrosswordSage is a typical example of applications where the GUI has been constructed through the use of a GUI editor, and little or no manual modification has been made to the generated code. This is actually apparent by examining the source code. In the source code for this application, I saw several generated comments from the GUI editor which verifies this is actually the case. Since the GUI for this application is purely constructed from a GUI editor, it means the complete main window view can be recovered statically using Swing Designer.

Java-Chess and jMSN are also examples of applications that are created by using a GUI editor. However, after the creation of the initial GUI design, the design was manually tailored to introduce abstraction. Furthermore, the initial GUI design is also refined by using the developer’s hand-written code, which introduces some complex program logic, for example the painting for the chess board. Some of the larger examples, such as, GanttProject and Freemind are good examples where the GUI code is completely implemented by GUI developers.

For these four applications, we can see that the design model was less than 50% of the size of the dynamic model. So we can see that the opportunities for view-based navigation are limited in the current approach. My approach captures the complete dynamic model; additional screen shots like the ones shown for Java-Chess are made available in Appendix A. This is to be expected since a dynamic analysis has complete runtime information.

It is important to note that the traditional GUI editor was not designed to work on arbitrarily tangled code bases. So the percentages in the table really tell us more about the structure of those code bases than about the editor itself. I feel the numbers provide some evidence that larger code bases tend to introduce more complexities. This could make the GUI hard to maintain without good tool support.

In some software development situations, smaller models might actually be better because they provide a useful abstraction. So, I clarify that in all these cases the static design model is simply a truncated version of the dynamic model, cut off at some levels in the tree. The results in the table describe how much truncation occurs. The truncation is apparent visually for Java-Chess in Figure 3.2 and truncation is illustrated in Figure 3.26.

These measurements provide a basis for judging the usefulness of view-navigation on a dynamic view. Furthermore, these measurements also show that there is not much utility gained by using a
traditional GUI editor for some complex GUI code. So, a developer unfamiliar with the source may find that the dynamic model is helpful to aid them to identify the mapping relationships between the rendered widgets and actual implementation. Basically, without my tool support a developer might find that it is difficult to identify the mapping relationships. However, these measurements do not provide a good comparison for judging the view-editing capabilities of my approach, since much of the information in the dynamic model is not static design, so I explore this next.

### 3.6.3 View-Editing Comparison

Table 3.7: Size of design information displayed by original and augmented GUI editor (FreezeFrame) compared to total amount of information in the dynamic view

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Original editor</th>
<th>FreezeFrame</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CrosswordSage</td>
<td>100%</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>Java-Chess</td>
<td>45%</td>
<td>56%</td>
<td>1.2x</td>
</tr>
<tr>
<td>jMSN</td>
<td>34%</td>
<td>80%</td>
<td>2.4x</td>
</tr>
<tr>
<td>GanttProject</td>
<td>8%</td>
<td>21%</td>
<td>2.6x</td>
</tr>
<tr>
<td>FreeMind</td>
<td>10%</td>
<td>53%</td>
<td>5.3x</td>
</tr>
</tbody>
</table>

Finally, the last metric measures the ratio of design information that is statically editable using the original editor versus the same editor augmented with my analysis (FreezeFrame). These results are displayed in Table 3.7.

In the second column we see the amount of statically editable widgets and widget properties for the original GUI editor. This is the same data as shown in Table 3.6 because all of the widgets and widget properties displayed by standard editors are statically editable. The third column measures the amount of statically editable information using the augmented editor, FreezeFrame. This was determined by intersection of static and dynamic models as described in Section 3.4 by following my editing constraint.

In the third column we see the ratio of editable design information using my approach versus the standard approach. The additional information appears because the dynamic analysis has “re-
attached” truncated sub-trees so they are reachable from the root main window. My approach didn’t “create” this new design information, I simply have a way of locating it and placing it in context.

The important consideration here is to judge whether the improvement I have made in these cases is of practical significance. For the experiment I have done, 3 out of 5 of the applications show at least 2x more design information editable in the context of the main window. Essentially, if a developer would like to change any of this additional information it can now be from a single conceptual module. Furthermore, the two larger applications appear to benefit the most. This gives evidence that my approach can help improve view-based editing even as application sizes scale and complexities are introduced.

In this chapter, I present a dynamic GUI editor called FreezeFrame. GUI developers can use this augmented editor to solve the first defect of traditional GUI editors: limited reverse-engineering capabilities. In the next chapter, I present a possible solution for GUI developers to solve the second defect of traditional GUI editors: no explicit support for coordinated behavior.

3.7 Discussion

3.7.1 Application

FreezeFrame could be used to assist GUI developers in understanding the UI-code mapping relationship between what they see on the screen and what is implemented in the source code. The static design and dynamic design of a user interface are often tangled together in the user interface implementation. Traditional GUI editors cannot reverse-engineer a complete design view for the user interface because of their limited reverse-engineering capabilities. FreezeFrame addressed the limitation by using a dynamic analysis approach to recover a complete design view for a GUI developer. Based on this complete design view, the GUI developer is able to navigate and edit more GUI designs.

3.7.2 Manual Effort and Automation

For a GUI developer who wants to use FreezeFrame to understand the UI-code mapping relationship between a user interface and the corresponding implementation, they will need to manually choose the user interface of interest. FreezeFrame automatically creates the UI-code mapping relationship. Next, I discuss these manual effort and automation in details.
Manual Effort

- The developer needs to run Aspect Weaver for the selected application to weave the aspects into the application.
- The developer needs to run the application and browse different user interfaces until they find the one of interest.
- The developer needs to notify FreezeFrame to make a “snapshot” for the user interface.

Automation

- The UI-code mapping relationship between the user interface of interest and the corresponding implementation is recorded and provided to the GUI developer in a WYSIWYG way.

From the above description of the manual effort and automation, we can see that the procedures of the manual effort have the potential to be naturally embedded into developers’ development or maintenance work flow (e.g. running the application and browsing different user interfaces till the developer finds the one of interest). However the UI-code mapping relationship between the user interface and the corresponding implementation is not always clear and traditional GUI editors have very limited support for finding the mapping relationship. FreezeFrame provides this mapping relationship to developers.

3.7.3 Scope

So far, FreezeFrame only provides support for the user interfaces that are implemented using the Swing GUI library. It cannot assist developers in understanding the user interfaces that are implemented using any other GUI libraries, for example, SWT or OpenGL. The user interfaces that are constructed by using some GUI libraries (e.g., SWT) are also structured GUIs. Providing support for these kinds of GUI libraries is very straightforward and only requires minor modification on my tool. However, some GUI libraries (e.g. OpenGL) use low-level API such as pixel-level operations to construct user interfaces. My research cannot be simply modified to adapt these kinds of GUI libraries because the user interfaces that are built by these GUI libraries are not structured GUIs. For a GUI that is not well structured, a UI-code mapping would be much more difficult or even impossible to create.
3.7.4 Limitations

FreezeFrame has several limitations. One limitation is that FreezeFrame uses a developer’s guided analysis to locate the user interface of interest, and then the dynamic model of the user interface can be captured by FreezeFrame’s dynamic analysis. However, there are some situations in which GUI developers are interested in a very specific user interface, which can only be triggered during the execution of a very particular execution path. Sometimes, this user interface is very difficult or even impossible to obtain during the normal application execution, for example, a user interface that is only presented when the application crashes. For dealing with this problem, some automated testing research such as [64] can be used to obtain coverage of GUI applications.

Another limitation is that a developer has to re-execute an application whenever any modifications have been done on a user interface through directly modifying the corresponding source code. The reason is that FreezeFrame uses a dynamic analysis approach and it can only create the UI-code mapping relationship during runtime. So the developer has to execute the user interface another time and find the mapping relationship between the modified dynamic view and the new code that they just added. In this case, a re-executing will be required every time when the GUI code has been manually modified. Next, the relationship between the modified user interfaces and the corresponding implementation can be recorded at runtime and presented to the developer through a WYSIWYG approach.

3.7.5 Threats to Validity

This subsection describes different threats to the validity of my quantitative evaluation. First, I describe the threats to external validity, which explains how I might be wrong in making a generalization of my approach. The issue concerning the external validity of quantitative evaluation is that the measurement is based on the main window because the main window scenario is easy to be determined for all the selected applications. However, some GUI applications might not have really complicated main windows. In this case, measuring other UI views would be a good alternative.

Another issue concerning the external validity of my quantitative evaluation is that I measured the number of classes containing GUI join points captured by GUIAspect for rendering of the main window view in my research. The reason is that a larger number of classes indicate more difficulty performing maintenance on the view without proper tool support. However, developers may not think this is true because they might consider the GUI code size problem creating the complexity for them to maintain the code. Since the code size of some applications I investigated are fairly small, for
example, CrosswordSage and Java-Chess, it would be interesting to look into other open source GUI applications that have large code sizes.

Second, I describe the threats to internal validity, which explains how I might be wrong in following the principle of cause and effect [73]. With respect to internal validity, the only problem that would arise is if somehow the GUI was manipulated without being intercepted by GUIAspect. This was not a large problem in my case since Java libraries strictly follow the conventions strictly set out by an industry standard. I have carefully inspected the visual appearance for all cases. In each case there are less than three small visual differences\textsuperscript{11}, for example, a difference could be caused by the fact that I have not yet implemented a pointcut for catching when widgets are disabled (i.e. “grayed out”).

\textsuperscript{11} These differences will be presented in Appendix A.
Chapter 4

Using Models to Explore GUI Behavior Implementation from Dynamic Views

As discussed in Chapter 3, my research assists GUI developers in maintaining and understanding non-animated user interfaces through a dynamic GUI editor. The dynamic GUI editor enhances the view-based navigation feature and the view-based editing feature of a traditional GUI editor, through a dynamic model. This model maintains a mapping between runtime widgets on screen and the actual implementation of the widgets. By using this mapping, a GUI developer canbridge what they see on the screen and what is implemented in the source code. This mapping is useful based on the evaluations that have been done in Section 3.6. However, this mapping is still not sufficient in assisting GUI developers with solving another problem.

Some parts of a user interface are stateful and reactive. They could be transformed to different states based on a user’s interactions. Recalling the example in Section 1.2.2, the appearance and behavior of the Catalog Browser page vary over time, due to mutations of the states made from the GUI statements in the code. A developer who wants to map a coordinated GUI behavior to the source code that implements this behavior has to find all the GUI statements that control the transitions of the states; furthermore, these transitions sometimes happen in a very short time. In this case, when the developer looks at the GUI behavior, it could be difficult for them to determine precisely the source code that implements a particular GUI state transition.

Currently, there are no existing GUI editors that resolve the above problem for GUI development and maintenance, because of the static nature of traditional GUI editors. Trying to solve the problem
requires GUI developers to use an ad hoc approach to understand a large amount of GUI code that can be scattered throughout different software modules.

In my research, I consider this problem the second defect of traditional GUI editors: they do not provide explicit support for coordinated behavior. To deal with this defect of traditional GUI editors, I want to help developers in navigating directly to the code responsible for controlling the dynamic GUI effects and animations. In my research, I am looking for an interactive approach to bridge the gap between a coordinated GUI behavior and a set of actual GUI statements that implement the behavior. To distinguish from view-based navigation, I call this behavior-based navigation. This is motivated by the fact that the behavior of a UI is usually easy to understand and semantically meaningful, unlike the implementation code. I want to help front-end developers use the live UI as an entry point into the lower-level implementation details.

The rest of this chapter is organized as follows: Section 4.1 presents the current techniques that are used to implement and understand GUI behaviors. Section 4.2 shows a motivating example. Section 4.3 presents technical details. Section 4.4 demonstrates some interactive widgets from a real Web application and presents two case studies to further explain the usefulness of my approach. Section 4.5 presents metrics from online sites. Section 4.6 provides discussion.

4.1 Implementing and Debugging GUI Behaviours

In this section, I describe the techniques that are used to implement GUI behaviours (Section 4.1.1) and how developers understand the implementation of GUI behaviours by using debugging techniques (Section 4.1.2).

4.1.1 Implementing GUI Behaviours

As described, a GUI behavior consists of a sequence of states. Therefore, to implement a GUI behavior, a developer must first write GUI statements to mutate the states and then “assemble” these statements in an appropriate temporal order to generate the GUI behavior. Here, I discuss the techniques that are normally used to implement a GUI behavior.

First, the GUI statements that mutate GUI states are often reused in different calling contexts that are executed under different situations to create different GUI behaviors because of the code abstraction mechanisms. For example, a window widget can have two different GUI behaviors, 12

---

12 It is important to note that my research is not trying to create a JavaScript debugging tool. Instead, I am creating a program comprehension tool. Debugging is used to help developers in understanding GUI behavior implementation. So I explain debugging here.
opening and closing. These two GUI behaviors could be implemented by using the same GUI statements in a function, but the function is executed in different calling contexts to control opening and closing animations.

Second, all the related GUI statements are organized together in an appropriate temporal order through some control flow statements. Executing all the GUI statements in the correct order is critical to creating a correct coordinated GUI behavior. For example, a window widget can display different text descriptions for different GUI behaviors such as opening and closing. In this case, the GUI statements that are responsible for opening the window have to be executed with the GUI statements that display the text “opening the window” instead of the text “closing the window”. This schema follows suit for closing the window.

4.1.2 Understanding Behavior Implementation by Using Debugging Techniques

Currently, a front-end developer normally uses debugging techniques to try to understand the implementation of GUI behaviors. By using these debugging techniques, a GUI application is first executed and then a program debugger is used to step through the code. A developer has to memorize all the places in the code where the state mutations happened. Obviously, this procedure is very complicated and error-prone, especially for the user interface that has very complex animations and state transitions. Next, I describe more details for this debugging procedure.

Traditional line-level debuggers do allow front-end developers to “step” through execution to identify all the GUI statements that they are interested in. However, using a debugger, a developer must first choose a breakpoint location from which to begin stepping through code. If this location is not immediately preceding the code that is responsible for the GUI effect, they will have to step through other possibly unrelated parts of the program. At each statement, the developer has to choose carefully whether to “step into” a particular function call or “step over” a function call. If the wrong choice is made, they could either waste more time or accidentally skip over the important code entirely. Furthermore, the developer still has to mentally organize all the statements they identified and manually distinguish different execution contexts for a particular GUI statement that has been executed at different times.

4.2 Motivating Example

In order to motivate this research, I use a case study of an existing open-source Web application called Java Pet Store 2.0 (henceforth, JPS 2) [87]. I chose JPS 2 because it is open-source, designed
as an interactive Web 2.0 application, and has been used by others as part of a case study [59]. JPS 2 is an open source project from Sun’s Java BluePrints [87]. It simulates a real-world pet store by providing features such as browsing, tagging, purchasing, and selling pets.

In this section, I use JPS 2 to illustrate the existing problems of GUI development and maintenance approaches for understanding a GUI behavior. In order for me to be able to describe some details of my study in depth, I focus on one particular page of the JPS 2, the Catalog Browser. This page offers the end-user several interactive widgets to control the application, as shown in Figure 4.1. The Catalog Browser page allows users to browse different pets that have been organized into different categories. For each particular pet, a user can see detailed information such as the pet’s name and description. Instead of listing all pets in a table and using hyper-linking to bring a user to different Web pages for a detailed description of the pets, JPS 2 uses a stylized UI to organize the pet’s information, as shown in Figure 4.1 and described in the following paragraphs.

Referring to Figure 4.1, users who use the Catalog Browser page can browse different pets that are organized into different categories. In the current implementation, all the pets are divided into five categories: Cats, Dogs, Birds, Reptiles, and Fish. Running down the left side of the Catalog Browser page is an accordion bar. This widget is a stylized tree-view for browsing categories of pets. All pets in each category are further divided into two sub-categories; for example, as shown in Figure 4.1, we can see that the Cats are further divided into the specific kinds of cats, such as Hairy Cat and Groomed Cat in this example.

The table rows for the categories interactively expand/deflate to reveal/hide sub-categories when the mouse cursor is positioned/removed from a category name. This “accordion” animation requires JavaScript programming to mutate the DOM attributes in a loop. Figure 4.1 shows the “Cats” row is expanded and the other categories remain deflated. When a user puts the mouse cursor on another category row, for example, Dogs, the Cats category row will be deflated to hide all the sub-categories for the Cats category. Interactively, the Dogs category row will be expanded to reveal all the sub-categories for the Dogs category. In the remaining part of this motivating example section, I use the accordion bar as an example to motivate my research by explaining some difficulties in mapping this accordion animation to the corresponding implementation.

Here we see the “Catalog Browser” page from which the end-user can browse prospective pets. This one page alone makes use of 2136 lines of code spread across 8 files. Consider the perspective of a front-end developer who would like to make changes to this Web page. If they were the original developer of all 2136 lines of code, they might be able to remember how all the code is mapped to

\[13\] This concept has been introduced in Section 2.1.
elements of the page and their behavior. However, this information is easy to forget since a single developer might be responsible for many Web pages from a single site and may even be working on several different sites over some time. If they were not the original developers, they may need a lot of time to understand the code.

Figure 4.1 [Example]: A snapshot for the “Catalog Browser” from Java Pet Store 2. An accordion bar on the left is marked with dotted lines. An expanded accordion row, “Cats”, is marked with solid lines.

In the original JPS 2, each accordion row is expanded and deflated at a constant speed. Consider a case where a front-end developer wishes to change to a different animation effect for these expanding and deflating animations. For example, when a row is expanding/deflating, the animation is accelerated at a decreasing/increasing rate. In this case, the developer who wants to modify the existing code has to understand how each individual effect (expanding or deflating) is implemented in the code. During the task, the developer is confronted with three problems.

First, they would have to determine which DOM nodes and which attributes of those nodes are responsible for the animation. This could be difficult because the implementation details could vary.
For example, the animation might involve any combination of style attributes such as \texttt{height}, \texttt{top}, and \texttt{clip}.

Second, suppose a developer figures out that \texttt{height} is the key to changing the animation. However, when they search through the code, there are two assignment statements to the \texttt{height} of some node in the JavaScript implementation. One of them is shown in Figure 4.2 and another one turns out not to be relevant. By looking at each statement individually, it is not always clear if the statement is relevant to the task at hand. They may also have to search the code to understand the calling context of each height setting statement, in other words, the function calls that lead to the statement’s execution.

![Figure 4.2](example.jpg)

\textbf{Figure 4.2 [Example]: The function \texttt{setHeight} on its own lacks the calling context that is needed to properly associate the function with the accordion bar animation.}

Third, suppose the developer determines the function as shown in Figure 4.2 contains the assignment statement they are interested in. In order to create the new acceleration/deceleration effect, they would want to change the argument value that was passed to a function call to \texttt{setHeight}, but not the definition of the \texttt{setHeight} code itself. So, on its own, the code in \texttt{setHeight} is not very useful for a developer. Now, when a developer searches \texttt{setHeight} in the code, they find two places where the \texttt{setHeight} function is called, as shown in Figure 4.3. Each one is relevant for the change task but for different reasons.

After taking an exhaustive amount of time to go through all these steps to explore the code, the developers may find that the first one (line 149) is involved with expanding an accordion row and the second (line 157) is involved with deflating it.

In the rest of this chapter, I seek to address the above three problems by allowing developers to interact directly with a runtime browser view. The developer executes the UI behavior that they want to modify or debug. A dynamic analysis creates a model to bridge the gap between the UI behavior and actual JavaScript implementation. Finally, the developer can use this model as an abstract high-
level pattern to help them in understanding the JavaScript implementation for the UI behavior they are interested in.

```javascript
147. if(...) {
148.   nHeight = nHeight + INCREMENT;
149.   divs[nExpandIndex].setHeight(nHeight);
150.   if(...) {
151.     if(...) {
152.       ...
153.     }
154.   } else {
155.     oHeight = oHeight - INCREMENT;
156.   }
157.   divs[oExpandIndex].setHeight(oHeight);
158. }
159. }
```

Figure 4.3 [Example]: Two function calls related to the accordion bar animation. The developer will need information to disambiguate the purpose of each function call. Some code is elided for illustration purposes.

### 4.3 Implementation Details

In my research, I want to help front-end developers in understanding the implementation of UI behaviours by using behaviour-based navigation. To achieve this behavior-based navigation, I propose using a similar approach to view-based navigation to assist developers in navigating their code exploration: a developer can easily navigate through the code by hyper-linking directly from a browser view. However, behaviour-based navigation is not as straightforward as view-based navigation, because a basic implementation of this approach is vulnerable to two problems.

First, mapping semantically meaningful events, such as the mutation of a visual attribute directly to a location in the source code (e.g., a statement) may not be helpful, because that one statement might be reused for several different purposes in the execution of the script. For example, informing a developer that an attribute was changed in a “setter” method for that attribute provides little useful
information. The “setter” method could be called many times in the execution of a script, in different contexts, for a variety of different purposes. In this research, I refer to this problem as the context-sensitivity problem.

For this reason, I have investigated the use of context-sensitivity to help provide a mapping. A context-sensitive approach captures not only the execution of individual statements, but also the state of the call stack, which can help distinguish among multiple executions of the same statement. I describe this context-sensitivity in Section 4.3.1.

Second, the visual behavior of a Web page (e.g., the way widgets are animated) is often achieved by a set of coordinated DOM attribute mutations. For example, a button’s appearance may change to reflect that the button is active when a panel is closed and change again to reflect it is inactive when the panel is open. The changes to the button appearance and panel appearance have a causal relationship. If a developer wants to change the widget animation, they may have to make coordinated changes to several DOM attributes. In this research, I refer to this problem as the coordinated behavior understanding problem.

For this reason, I have investigated the use of a custom control-flow model, the DOM Mutation Graph (DMG), that developers can use to map causal relationships, which have been seen in the browser view, to script source code. The DMG model for UI execution history is presented in Section 4.3.2.

In my research, I created a prototype tool called ScriptInsight [57][36]. This tool is designed to help front-end developers interactively in exploring the relationships between elements in the browser view and script code. The tool is implemented as a JavaScript front-end, to execute within a standard Web browser, and a separate HTTP proxy executable. In Section 4.3.3, I describe the runtime tracing infrastructure that is used by my prototype. In Section 4.3.4, I describe the procedure that GUI developers should follow when they use this tool. To demonstrate my approach in using DMG to explore script code, I present case studies of JPS 2 (Section 4.4) and metrics taken from several popular online sites (Section 4.5).

### 4.3.1 DOM Mutator and DOM Mutator Context

In this subsection, I describe two basic concepts that are used by my tool, DOM Mutator and DOM Mutator Context. These two concepts represent a subset of the JavaScript statements executed and the corresponding calling contexts of the statements, which are monitored by my tool during the program execution. In the remaining part of this subsection, I provide definitions and examples for these two concepts.
Def 4.1: DOM Mutator
A DOM mutator is a JavaScript statement that mutates the state of a DOM. This can be either by directly setting an attribute of a node\textsuperscript{14} or through any one of the functions in the DOM standard\textsuperscript{15} [100].

For example, in JPS 2, the \texttt{height} attribute of some nodes is mutated dynamically. My tool records this fact so that a developer concerned with an animation related to the \texttt{height} can quickly locate the corresponding implementation. In many cases, pages are built from a collection of JavaScript files. For this reason, each statement is located by a line number and the URI of a JavaScript file or HTML document where the DOM statement occurs.

In many cases, dynamic information is needed to distinguish the calling context in which some statement executed. For this reason, my tool captures the calling context of each DOM mutator execution instance, which is called the \textit{DOM Mutator Context}.

Def 4.2: DOM Mutator Context
The DOM Mutator Context is an ordered list containing the location of all JavaScript function calls active at the moment of execution for some DOM mutator. This context captures the stack of function calls from some event handler invoked by the browser, to the statement.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Example: The DOM mutator context of the \texttt{height} property mutator for the accordion row example, as described in Section 4.2.}
\end{figure}

Figure 4.4 presents the DOM mutator context of the \texttt{height} property mutator for the accordion row example, as described in Section 4.2. By looking at this calling context, a developer can quickly

\begin{flushright}
\textsuperscript{14} e.g., node.id = ‘submit’ \\
\textsuperscript{15} e.g., node.appendChild(\ldots)
\end{flushright}
identify the function that requires changes to achieve the modification for the accordion row animation.

Figure 4.4 provides a simple example for the usage of the calling context information. Next, I provide a more complicated case. Consider an example from eBay where JavaScript library code is used to build “widgets.” These widgets are an aggregation of DOM nodes that are encapsulated behind a high-level widget interface. Suppose a developer is interested in a particular instance of an eBay drop-down menu. They might wish to modify the parameters that were used in the construction of the menu. Using my tool, they could click on some part of the menu to be hyper-linked to the DOM mutator where that part of the menu was created. However, since these nodes were created as an internal part of the widget library, the developer would not want to actually change the library code but rather find where it was called from for this menu instance. In this case, by using the captured DOM mutator context, the developer can be directed to code where a call to the library was made. From this point they can change the parameters to the widget library rather than the internals of the widget library itself.

4.3.2 DOM Mutation Graph

To help developers review mutations in an animation that occurs over some time, I have to consider the history of DOM mutations related to each DOM node attribute. My tracing infrastructure captures a complete trace of all DOM mutator contexts, including the value (e.g., 10, ‘red’, ‘http://..’) that is assigned by the mutator for each context. This complete history trace is useful for developers because they can review the state changes that are made to a property over time.

Each complete history trace is a representation of the execution history for a specific instance of a JavaScript event handler (e.g., onclick, onhover). This execution history captures all mutations made during the activation of the handler (i.e., while the handler is on the call stack). I use this partitioning of trace information because each particular event handler is commonly responsible for creating one particular animation or dynamic effect on the page. Scoping the generation of models to align with event handlers allows a developer to focus on a particular animation or effect, and the way it may affect multiple attributes of multiple DOM nodes, in a coordinated fashion. Next, I provide a definition for the complete history trace (henceforth, just trace).

**Def 4.3: Trace**

The trace \( t_c \) can be presented as \( C_1, C_2, ..., C_n \). Each element \( C_i \) in \( t_c \) is a DOM mutator context. This trace represents the execution order of different DOM mutator contexts.
However, it is well known that dynamic traces can sometimes overwhelm a user with a large magnitude of data, making the information not valuable. For this reason, I have designed an abstract trace mechanism.

**Def 4.4: Abstract Trace**

A abstract trace \texttt{atc} can be presented as \( S_1S_2 ... S_i^k ... S_n \). Each element \( S_i^k \) in the \texttt{atc} is a DOM mutator context, and \( k \) represents that \( S_i \) has been repeated \( k \) times. For any sub-trace of the \texttt{atc}, \( S_j = c_t c_{t+1} ... c_n \), there is no adjacent trace such as \( S_j \equiv S_{j+1} \).

The abstract trace mechanism works by abstracting the repeated patterns [49] of DOM mutator contexts. To create an abstract trace, I consider the calling context for each statement execution instance an element or “letter” of some alphabet. The sequence of entries identified by the trace is then considered a string or “word” over the alphabet. Using a well-known algorithm [49], I find all the repeating patterns in this string. Essentially, any substring \( a \) that is repeated \( i \) times is then represented as \( a^i \).

For example, consider the accordion bar height animation. The height’s initial value is set at one statement that has a calling context \( a \), and then repeatedly incremented in an event loop, at a statement that has a context \( b \). Finally, once the accordion is deflated it is repeatedly decremented at a statement that has a context \( c \).

A complete trace could include many entries that would make it hard to map the high-level behavior in the browser view to the code. By viewing an abstraction of the trace, \( ab^i c^j \) (where \( i \) and \( j \) are the number of repetitions for each loop respectively), a developer can intuitively map the semantics of the accordion animation to the code.

To visually present the abstract execution traces for developers, I designed a mechanism to represent JavaScript execution as a variation of a traditional control-flow model, which is described in the following definition.

**Def 4.5: DOM Mutation Graph**

\( \text{DMG} = \{V, E\} \). In a DMG, \( \text{FORALL } \{v_i\} \text{ IN } V : c_m = f(v_i) \rightarrow c_m \in \text{atc} \), and the function \( f() \) is bijective. So, there is a one-to-one mapping between each node in a DMG and each element in the \texttt{atc}. \( \text{FORALL } \{v_i, v_j\} \text{ IN } E : c_m = f(v_i) \& c_n = f(v_j) \rightarrow c_m c_n \in \text{atc} \) or there exists \( S_i^k = c_n ... c_m \ & \ k > 1 \).
In a DMG, each node corresponds to an element in the abstract trace. Note that each node in a DMG does not represent a DOM node or a DOM node property, it actually represents a DOM mutator context. The trace of concrete mutations, including the attribute values assigned, can be retrieved interactively by interrogating each node. Edges in the model correspond to the sequencing of statement execution. A directed edge is created from node \( u \) to node \( v \) if there is a trace entry for \( u \) followed by a trace entry for \( v \).

**Figure 4.5 [Example]: The DMG of the accordion row’s execution.**

For example, Figure 4.5 presents an example for the DMG of the accordion row’s execution, which is activated by an `onhover` event. In Table 4.1, the complete history trace (\( tc \)), the abstract trace (\( atc \)), and the DMG representation (\( DMG \)) of the accordion row behaviors are presented. This example will be further explored in Section 4.4.2.

**Table 4.1: The complete history trace (\( tc \)), the abstract trace (\( atc \)), and the DMG representation (\( DMG \)) of the accordion row behaviors.**

<table>
<thead>
<tr>
<th>( tc )</th>
<th>( height0, ..., height0, height1, ..., height1, height0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( atc )</td>
<td>( height0^0, height1^0, height0 )</td>
</tr>
<tr>
<td>( DMG )</td>
<td>( {[height0, height1]}, {height0 \rightarrow height0, height0 \rightarrow height1} )</td>
</tr>
</tbody>
</table>

My model is similar to traditional control-flow models, such as a control-flow graph or call-graph, in that each node represents some implementation-level artifact. However, I consider only the set of statements that affect the visual appearance of the UI and distinguish those statements based on dynamic context information. This allows the model to become a bridge between the flow of changes that a developer can see directly in the browser and the implementation that is causing those changes.

In summary, by examining a DMG, a developer can first obtain abstract high-level comprehension of how a coordinated behavior is implemented by a set of GUI statements. Next, the developer can quickly navigate to any of the statements. Upon being linked to a statement, they would then have
access to the statement execution context. In this way, both calling context and DMG can be used together in linking the Web browser UI view to implementation.

4.3.3 JavaScript Code Instrumentation and Tracing JavaScript Execution

In Web applications, the JavaScript code instrumentation is different from the code instrumentation approach that is used in Chapter 3. The reason is that the place where the code is actually executed and the place where the code is actually instrumented are not on the same machine. One possible solution is to instrument the code on the server side and then deliver the instrumented code to the client. However, this approach strongly restricts the application of my research to the projects in which developers have no control over the server side. In this case, the JavaScript Instrumentation Proxy [51] technique can be used to instrument JavaScript code by using a proxy. So, the server side JavaScript code is first delivered to the proxy, and then the JavaScript code is interpreted and injected with my tracing code. In this research, runtime tracing is implemented as a set of JavaScript functions that are called by tracing code injected into existing scripts. Scripts are intercepted and manipulated by a client-side HTTP proxy. I use the open-source Rhino [74] JavaScript compiler framework to convert scripts into an abstract syntax tree (AST), which is then transformed to add the tracing code. Finally, the instrumented code is delivered to the client’s browser and then executed. During the execution, the tracing code dynamically creates the behavior-implementation mapping.

4.3.4 Using ScriptInsight

A developer using my tool will install and point the browser to the HTTP proxy that provides instrumentation of existing JavaScript code. Using ScriptInsight, developers can switch the Web browser between normal execution mode and inspection mode. In script inspection mode, a developer can select a widget and trigger the widget’s associated events in the browser view. For example, the developer might select a particular image or table row they are interested in. Next, the developer is brought to a panel to choose an event that is associated with the widget and that the developer is interested in. Finally, they can use the previously recorded DMG that is generated by the selected event to guide their code exploration for a particular UI behavior.

\[\text{In my research, my tool only traces DOM mutators, corresponding mutator contexts and the execution order of different mutator contexts.}\]

\[\text{The widget that a developer is interested in might have more than one associated event; for example, an image widget can be associated with onclick event and onhover event. Different events often cause different UI behaviors that generate different DMGs.}\]
By selecting a node in the DMG, the developer is hyper-linked to the file for the associated JavaScript statement in a special text editor. In the editor, the cursor position is set for the line number of the statement, for convenience. This text editor includes a drop-down menu for the developer to navigate the calling context for a given statement execution.\(^ {18}\) This allows the developer to jump up and down the call stack as shown in Figure 4.6. This call stack information was captured precisely for that instance of statement execution in the trace history.

![Image]

**Figure 4.6 [Example]:** Selecting a function call location from the calling context. The (?) entry references an anonymous JavaScript event handler function. A mutation of the `style.height` attribute for some DOM node was made in the function `setHeight` that is shown at the top of call stack.

## 4.4 Catalog Browser Examples

I have proposed and implemented a novel JavaScript tracing approach to deal with the defect 2 of traditional GUI editors for Ajax applications. To understand the usefulness of the approach, I studied how my approach could be used on the existing JPS 2, particularly the Catalog Browser page, as described previously. Here I describe the Catalog Browser in detail, to explain how a front-end developer could make use of my approach.

\(^ {18}\) More details are provided in Section 4.4.2.
Figure 4.7 [Example]: A snapshot for the “Catalog Browser” from the JPS 2 Catalog Browser with markup to point out relevant interactive widgets described in the paragraphs of this section. An accordion bar on the left is marked with dotted lines. Label (A) is an expanded accordion row, “Cats”. Label (B) is a slider thumb, which is a thumb image for a particular pet. Label (C) is a scroll right button to roll the pets that are displayed in the scrollbar. Label (D) is a collapse button to control the raising and lowering of the information pane. The “Information Pane” (E) has been revealed, and all information for this pet is presented to users.

When a user chooses a particular sub-category, for example Groomed Cat, a set of cats that belong to this sub-category are loaded into a scrollbar that is at the bottom of the Catalog Browser Web page. Then, the user can press the two scroll buttons on the top left corner and the top right corner of this scrollbar (Figure 4.7 (C)) to roll left and right to different pets. Furthermore, when the user wants to see more details of a particular pet, they can click on the thumbnail picture (Figure 4.7 (B)) of the pet and more detailed information, such as name, price, and description, are loaded into an information pane (Figure 4.7 (E)) including a larger picture of the pet loaded into a pet view window (Figure 4.7 (F)). The user can then click on the collapse button (Figure 4.7 (D)) on the information pane to raise and drop the information pane. By doing this, users can choose to see or hide a more detailed description for the specific pet.
Several interactive widgets of the Catalog Browser page have been marked in Figure 4.7 and summarized in Table 4.2. These widgets show just some of the variety of techniques used in JavaScript programming for interactive Web applications. For applications like JPS 2, my research attempts to help developers in mapping the dynamics of the UI from the browser view to the code.

Table 4.2: Relevant HTML elements for the JPS 2 examples. The corresponding properties are dynamically mutated to create various effects described in this section. The notation (2) shows that the property has been mutated twice, and the (*) shows that the property has been mutated in a loop.

<table>
<thead>
<tr>
<th>Name</th>
<th>Tag Name</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Accordion Row</td>
<td>TR</td>
<td>height(*)</td>
</tr>
<tr>
<td>B Pet thumb</td>
<td>IMG</td>
<td>clip(<em>), left(</em>), visibility(2)</td>
</tr>
<tr>
<td>C Scroll button</td>
<td>IMG</td>
<td>visibility(2)</td>
</tr>
<tr>
<td>D Collapse button</td>
<td>IMG</td>
<td>src(2), title(2)</td>
</tr>
<tr>
<td>E Info Pane</td>
<td>DIV</td>
<td>clip(<em>), height(</em>), top(*)</td>
</tr>
<tr>
<td>F Pet View</td>
<td>IMG</td>
<td>src(<em>), opacity(</em>), visibility(2)</td>
</tr>
</tbody>
</table>

In the remaining part of this section, I first present a qualitative argument to scope the focus of the evaluation in Section 4.4.1. This argument will involve two particular examples from the Catalog Browser page: Accordion Row (Section 4.4.2) and Info Pane (Section 4.4.3). Through these two detailed case studies, I want to provide a justification for this argument. The two case studies are conducted to provide an in-depth understanding and relatively objective evaluation on ScriptInsight. The rest of the widgets of Table 4.2 are discussed in Section 4.4.4.

4.4.1 Qualitative Argument

In this subsection, I present a qualitative argument [30]. This argument is based on the comparisons to the prior development and maintaining approach that has widely been used for rich interactive Web applications. My argument is as follows:

- Defect 2 of traditional GUI editors (no explicit support for coordinated behavior) involves two major problems: the context-sensitivity problem and the coordinated behavior understanding problem.
• ScriptInsight enables two features for GUI developers: showing developers DOM mutator contexts for designated DOM mutators and the DMG used to visually present the execution order of related DOM mutator contexts.

• The context information provided by ScriptInsight can be used to address the context-sensitivity problem.

• The execution order information provided by ScriptInsight can be used to address the coordinated behavior understanding problem.

• By addressing the two problems through ScriptInsight, we can say that ScriptInsight provides an abstraction of the coordinated behavior between UI elements.

To prove the above argument, I need to answer the following two research questions,

**RQ 1:** How can the context information provided by ScriptInsight be useful in addressing the context-sensitivity problem?

**RQ 2:** How can the execution order information provided by ScriptInsight be useful in addressing the coordinated behavior understanding problem?

The two case studies I will evaluate represent two common situations. Each case is focused on answering one research question above, i.e., one for evaluating the usefulness of the context information and the other for evaluating the usefulness of the execution order information.

The first case study in Section 4.4.2 is the Accordion Bar example, which has been highlighted in Figure 4.7 (Tag A). In this case study, a developer is assigned a code modification task to modify the expanding and deflating effects of the accordion bar. I will use this case study to evaluate the usefulness of the calling context to address the context-sensitivity problem and answer research question 1.

The second case study in Section 4.4.3 is about the “Info Pane” and “Collapse Button” as shown in Figure 4.7 (Tag E) and Figure 4.7 (Tag D), respectively. As we know from Section 1.2.2, there is a causal relationship between the information pane and the collapse button. A front-end developer is assigned a code modification task to change the height of the maximized information pane because some pets have longer descriptions and the current height of the information pane does not provide enough space for those descriptions. Therefore, the descriptions are cropped at the bottom and cannot
be fully displayed. In this case, developers have to understand a set of attribute mutations of the information pane and collapse button, which are performed in coordination. I will use this case study to evaluate the usefulness of the DMG model to address the coordinated behavior understanding problem and answer research question 2.

For each case study, I first introduce the behavior of the widget at a high level from the perspective of an end-user. Then, I will provide a lower-level description for the widget from a front-end developer’s perspective, to reveal the challenge a developer faces while exploring the source code to complete a maintenance task in a traditional approach. Finally, I will describe the model that is generated by using my tool to demonstrate how my research can be used to bridge these two different perspectives. A developer can use this model as linked from the browser view to quickly get into the JavaScript programming details to facilitate program maintenance or understanding. By comparing two different scenarios, for example, the code modification task fulfilled by using a traditional approach with a well-known Web developing tool Firebug[32], or with the assistance of my approach and tool ScriptInsight, I answer the two research questions and provide justification for my argument.

4.4.2 Case Study: Accordion Bar

The first case study is the accordion row example (Figure 4.7, Tag A), described in Section 4.2. In this case study, a developer wants to modify the expanding and deflecting animation of the accordion row. Instead of expanding/deflecting at a constant speed, the developer wants the expanding/deflecting animation to accelerate at a decreasing/increasing rate when a row is expanding or deflecting. I use this case study to evaluate the usefulness of the context-sensitive approach and answer research question 1. Here, I first demonstrate how a developer could complete this code modification task by using a traditional approach with the assistance of Firebug. Then I demonstrate how the same code modification task can be done through ScriptInsight.

4.4.2.1 Using a Traditional Approach

A developer who wants to solve this code modification task using a traditional approach has to solve the three difficulties that have been described in Section 4.2. Here I will provide more details to explain how a front-end developer solves the three difficulties.

The first difficulty is easily solved by using Firebug. As shown in Figure 4.8, the developer can visually inspect the runtime Web page and select the accordion row, and then they can see the corresponding HTML element for this accordion row. From the highlighted HTML in Figure 4.8, the developer can easily find that the style attribute height is the key for this accordion row widget. For
example, when the developer puts the mouse cursor onto an accordion row that is expanding/deflating, they will see that the value of the \texttt{height} is increased/decreased continuously until the maximized/minimized height of this accordion row is reached.

Figure 4.8 [Example]: A front-end developer can use Firebug to easily find that \texttt{height} property is the key property of the accordion row animation. A developer positions the mouse cursor on the expanded accordion row “Cats” with Firebug’s inspect mode turned on, they will see the corresponding HTML elements that create the accordion row. When the developer puts the cursor onto an accordion row that is expanding/deflating, they will see the value of the \texttt{height} property is increased/decreased continually. In the current snapshot, the value of the \texttt{height} property is 120px, which reflects the open status of this accordion row.
After finding the key attribute for this accordion row, the developer still has to find the JavaScript code that implements the behavior of this accordion row, i.e., how the `height` attribute is mutated in the JavaScript code. In this scenario, Firebug does not provide any assistance for this functionality. Therefore, developers cannot depend on Firebug to solve the other two difficulties.

To solve the second difficulty, the developer has to search the code and identify any code snippets that contain the keyword “`height`” from the 8 related files of the accordion row. The developer has to understand all the related code snippets either by themselves or by consulting some senior project developers. The developer’s efficiency in solving this difficulty is solely dependent on how well the implementation of the project has been organized by the previous developers. Furthermore, this difficulty could be exacerbated by the fact that the code snippets that the developer found sometimes provide limited information, which leads to the third difficulty, the context-sensitivity problem.

The front-end developer who finds a code snippet that contains the keyword “`height`” normally attempts to understand the function that contains the code snippet because the code snippet itself normally does not provide a complete context for the developer to understand the program logic. However, trying to understand the related function is also not sufficient because of the code abstraction mechanisms. This function could be called in different contexts, and different parameters could be passed into this function, which create completely different behaviors for the widget. A front-end developer who wants to modify a specific behavior for the widget has to understand all the calling contexts and the parameters that are passed into this function. In the worst case, a front-end developer has to understand all the code in the project.

From the above description of how a developer attempts to solve the three difficulties, we can see that a developer can use Firebug to solve the first difficulty, but the second and third difficulties still cannot be solved. In the remaining part of this subsection, I describe how a developer solves the same code modification task through my approach.

### 4.4.2.2 Using ScriptInsight

Using my proposed approach, a developer could have chosen to see a model of the accordion row’s execution. The model generated by my tool is shown in Figure 4.9. In the model, each node represents a statement that mutated some visual DOM attribute and the calling context in which that statement executed.

From the model, a developer could determine that the animation was created by alternating, repeated executions of the context represented by `height0`, followed by repeated executions of the context represented by `height1`. 
Figure 4.9 [Example]: The abstract behavior of an accordion row presented as a DMG. The two traces of the height values (overlayed on the model with block arrows, in the figure simply for illustration) show the information displayed to a developer when selecting one of the two nodes in the model.

```javascript
Row.prototype.setHeight = function(nH) {
    this.h = nH;
    this.div.style.height = nH + "px";
}

Row.prototype.getTotalHeight = function() {
    return this.div.offsetHeight;
}

Row.prototype.getHeight = function() {
    return this.h;
}

this.handleEvent = function(args) {

```

Figure 4.10 [Example]: Selecting a function call location from the calling context. The (?) entry references an anonymous JavaScript event handler function. A mutation of the `style.height` attribute for some DOM node was made in the function `setHeight` shown at the top of the call stack. This mutation corresponds to the `height0` node from Figure 4.9. The stack contents serve to distinguish this execution of `setHeight` from those corresponding to node `height1`.

By selecting each node in my tool, the developer can perform two functions. First, the developer can view a trace of the values that were set in each context as shown in Figure 4.9. From the trace, it is clear which one is responsible for expanding and which one is responsible for deflating. Having the
information in mind, the developer can hyper-link to the corresponding source for the one they are interested in. In the model, height0 links to the executions of Figure 4.2, which were made from line 149 in Figure 4.3; height1 links to the following repeated executions of Figure 4.2, which were made from line 157 in Figure 4.3.

This calling context information is presented to front-end developers who use my tool through a text editor, as shown in Figure 4.10. For example, when the developer clicks on height0, they will be brought to a JavaScript text editor as shown in Figure 4.10, which shows the calling context information of the setting height statement for the accordion row’s expanding effect.

Now, the developer can find the correct locations to change argument values for each call, to implement the desired acceleration/deceleration change. In Section 4.4.3, I describe another case study regarding using a DMG for exploring JavaScript code.

4.4.2.3 Conclusion

A GUI developer who wants to change the animations of the accordion row has to find the related DOM mutator that mutates the height attribute of the accordion row widget. Because of the context-sensitivity problem, only finding the DOM mutator is not sufficient. In this case, the developer needs an effective way to identify the appropriate context information for different GUI effects. Using this information, they could change the parameters of the appropriate function calls to change the animations of the accordion row. Using a traditional approach, the developer has to manually search the code, understand it, and distinguish different calling contexts for the expanding and deflating effects. Using ScriptInsight, the developer is directly navigated to the related DOM mutator, and then the corresponding context information is provided to the developer through a drop-down menu. Next, the developer can use this context information to quickly locate the appropriate context (the function call) that they are interested in. Finally, by changing the parameter that is passed into that function call, the animation of the accordion row can be changed. Comparing the two scenarios of using a traditional approach or ScriptInsight, I answered the research question 1 by concluding that DOM mutator context provided by ScriptInsight is useful to address the context-sensitivity problem.

4.4.3 Case Study: Information Pane and Collapse Button

In order for me to be able to demonstrate the usefulness of the DMG model, I chose to use the “Info Pane” (Tag E) and “Collapse Button” (Tag D) on the Catalog Browser page of JPS 2 in Figure 4.7. The reason that I chose to use these two widgets is that they have some causal relationships, which
requires a set of DOM mutators executed in an appropriate temporal sequence. In this case study, I want to answer research question 2 by presenting the importance of showing front-end developers this sequence in an abstract high-level model through a DMG.

The information pane (Tag E) describes the detailed information for a selected pet (e.g., name, description, and rating). The collapse button controls the raising and lowering of the information pane. When the information pane is maximized, it partially obscures a pet image. When the pane is minimized, it appears to slide behind the scrollbar. The information pane widget is mapped to a div element in the DOM. This information pane can be expanded upward and compressed. This animation is performed by mutating clip, height, and top attributes in coordination.

The collapse button (Tag D) controls the raising and lowering of the information pane. It is an img element in the DOM. Two places in the code set the src attribute of this element. The collapse button’s icon is changed to a down arrow when the information pane becomes fully raised and changed to an up arrow when the information pane becomes fully lowered. Furthermore, there are two places in the code to set the title attribute of this element: when the information pane is minimized, title is set to “Show Details”, and when the information pane is maximized, title is set to “Show Less Details”. Therefore, when a user hovers the mouse cursor on the collapse button, the corresponding text will be presented.

Suppose a developer is assigned a code modification task to change the height of the expanded information pane because some pets have longer descriptions and the current height of the information pane does not provide enough space for those descriptions. In those cases, the descriptions are cropped at the bottom and cannot be fully displayed. In the rest of this subsection, I describe two scenarios for this code modification task. One scenario describes how this code modification task is done by using Firebug. Another scenario presents how this code modification task is fulfilled through ScriptInsight.

4.4.3.1 Using a Traditional Approach

For a front-end developer who wants to fulfill this code modification task through the assistance of Firebug, the first step is to find the actual DOM element that represents the information pane. This is easy to achieve. Using Firebug, a front-end developer can point the mouse cursor onto the area of interest, and then the corresponding HTML element is highlighted. As shown in Figure 4.11, when a developer points to the information pane widget on the dynamic runtime view, the corresponding HTML element is discovered, which is a <div> tag with the id attribute “infopane”.

After this DOM element is found, the developer has to start looking for the code snippets that contain the string “infopane” because Firebug does not connect the DOM element with the
JavaScript code that mutates the DOM element. Firebug provides very limited assistance for this code-searching task.

Figure 4.11 [Example]: A front-end developer uses Firebug to find the HTML element of the information pane widget. Using the inspection of Firebug, the developer is able to identify all the important properties of the information pane. When the developer positions the cursor on the information pane during the sliding up/down of this information pane, they will see the values of the identified properties increasing/decreasing respectively.

Finding the corresponding code is not easy because of the GUI code size problem and GUI code scattering problem that have been discussed in Chapter 1. For this catalog page, there are 8 files and 2136 lines of code. A developer who is assigned this task may first search the keyword “infopane” in
all the related files. After searching, the developer identifies four files that contain this keyword: `catalog.jsp`, `catalog.js`, `scroller.css` and `scroller.js`.

The developer first examines the `catalog.jsp` file. In this file, they find the HTML elements that create this information pane as shown in Figure 4.12.

```
1. <div id="infopane" class="infopane">
2.   <table class="infopaneTable">
3.     <tr>
4.       <td id="infopaneName" class="infopaneTitle"> </td>
5.       <td id="infopaneRating" class="infopaneRating"> </td>
6.       ...  
7.       <td id="infopaneDetailsIcon"> </td>
8.       <td id="infopaneDescription" colspan="6" class="infopaneDescription "></td>
9.     </tr>
10.  </table>
11. </div>
```

**Figure 4.12 [Example]:** The code snippet from `catalog.jsp`. This code displays the HTML elements that create the information pane. Some code is elided for illustration purposes.

However, finding the creation of the info pane does not help this code modification task, because the developer is interested in the JavaScript code that mutates this info pane to create the sliding up animation. By understanding the sliding up animation, they will be able to find the code that controls the predefined maximum height that the info pane can reach.

Suppose that, after spending a substantial amount of time understanding the four related files, the developer can then identify the file `scroller.js` as the controller function for the info pane slide. A keyword “infopane” search in this file will identify two functions that appear most related: `maxmizeInfoPane` and `minimizeInfoPane`. The names of the function here provide some useful information on the functionality for the developer. In this case, the developer can easily find the appropriate function to look into, i.e., `maxmizeInfoPane`. However, in some situations, functions do not have meaningful names. Therefore, the developer has to try to understand the implementation of the functions.
When the sliding upward action is fired, this `maximizeInfoPane` function is called periodically to change the values for `style.height`, `style.top` and `style.clip` to create the expanding animation. The incrementing of the values only stops when the predefined boundary condition is achieved: “infoPaneLoop < INFOPANE_EXPAND_HEIGHT”. Therefore, this code-modification task can be fulfilled by changing the value of the `INFOPANE_EXPAND_HEIGHT` variable to a larger number.

From the above description, we can see that the most tedious part of this modification job is to find how the information pane is animated from the minimized state to the maximized state in the JavaScript code. This animation is more difficult to map to the corresponding implementation than to find where the info pane is created. In the rest of this subsection, I present how to use ScriptInsight to assist with this need. I demonstrate how the same code modification task can be done through my approach.

4.4.3.2 Using ScriptInsight

Suppose a GUI developer wants to use ScriptInsight to solve the code modification task. They could go through four steps to completely understand the implementation for the UI behaviors of the information pane and collapse button. Next, I describe the four steps in detail.

**The First Step**

When the developer first browses the Catalog Browser page, the initial state of the Info Pane is minimized. As shown in Figure 4.13, from my tool, the developer sees an empty table for the DOM mutator context and the corresponding trace values. The reason is that the developer does not click on the Collapse Button and the sliding upward animation is not executed yet. Therefore, there is no tracing recorded by ScriptInsight. Furthermore, the corresponding DMG is only presented with an arrow (arrow 1.1), as shown in Figure 4.13, to indicate the start of the DMG.

**The Second Step**

When the collapse button is clicked the first time, the information pane is raised with sliding animation until the predefined maximum height is achieved. My tool records three mutated attributes of the information pane and two mutated attributes of the collapse button, as presented in Figure 4.14 (b). In (b), the various contexts in which attributes of the information pane and collapse button are mutated. The trace information of value changes associated with each context is shown in the second column (some are elided for illustration). Note that as is common, the coordinate for `top0` is measured as the pixel distance from the top of the screen; hence, it is decreasing.
Figure 4.13 [Example]: Step 1: the initial UI for the Catalog Browser without any user interactions (a). In this case, there are no trace values because the tracing is not fired yet (b). Accordingly, the DMG is empty; only an arrow (1.1) is presented to indicate the start of the DMG (c).

From Figure 4.14 (c), we can see that the nodes can be divided into two sets. The first set contains all the nodes related to the raising animation of the information pane: height0, top0, and clip0. The mutations of the three different attributes, style.height, style.top, style.clip, have been executed continuously in an event-loop, shown by the recursive edges (arrow 2.1, arrow 2.2, and arrow 2.3) out of clip0 and height0. The second set contains two nodes related to the animation of the collapse button: src0 and title0. The src0 node represents the DOM mutator context that is responsible for setting the image to the down arrow when the info pane is fully expanded (indicating this button can be clicked to compress the info pane). The title0

---

19 The actual information pane attributes in the code are style.height, style.top, and style.clip. Here, I used height0, top0, and style0 to represent the corresponding DOM mutator contexts for the three attributes.
node corresponds to the DOM mutator context that sets the `style.text` attribute to “Show Less Details”.

![DOM Mutation Graph](image)

**Figure 4.14** [Example]: Step 2: the UI for the Catalog Browser when the Info Pane is fully expanded (a). During the expanding, my tool captures the various contexts in which attributes of the information pane and collapse button are mutated, and the trace information of value changes associated with each context are shown in the second column (some are elided for illustration) (b). The flow of the information pane and collapse button presented as a DMG is shown in (c). Each node corresponds to the entries from the table as shown in (b).

Using my tool, the developer only needs to finish this second step and then will immediately realize that the `height0`, `top0`, and `clip0` nodes are responsible for the raising animation of the
information pane. By exploring any of these context nodes, they can easily find the `maximizeInfoPane` function by using the calling context information that is provided by ScriptInsight.

![Image of a browser window with a pet information pane open](image.png)

**Figure 4.15 [Example]:** Step 3: the UI for the Catalog Browser when the Info Pane is fully compressed (a). During the compressing, my tool captures all the newly executed DOM mutate contexts and associated values as shown in (b). Compared to Figure 4.14 (b), the newly captured DOM mutator contexts contain all the contexts that are executed to create the compressing animation. The corresponding DMG is evolved as shown in (c).
To solve this particular code modification task, a partial DMG as shown in Figure 4.14 is sufficient for the developer to identify the key function that controls the raising animation of the information pane. By finding this function, the task can be fulfilled. However, if the developer is assigned a different code modification task to change the height of the minimized information pane, the developer has to have a more complete DMG to help them in understanding how the deflating animation is implemented. In this case, the developer must further interact with the collapse button to fire the deflating animation of the information pane to obtain a more complete DMG. In the remaining part of this subsection, I illustrate two more steps to create a complete DMG for the information pane and collapse button animation.

**The Third Step**

In the next step, the collapse button is clicked again, and the information pane is compressed with sliding downward animation. The DMG is evolved with another set of nodes that reflects to this lowering effect, as shown in Figure 4.15 (c). Furthermore, the table in Figure 4.15 (b) is evolved with all the DOM mutator contexts that are executed during this information pane lowering and collapse button switching animation. The trace information of value changes associated with each context is also presented. Similar to Figure 4.14 (c), the evolved DMG also has a circle for nodes height1, top1, and clip1. The reason is that these three contexts are executed repeatedly (arrow 3.2, arrow 3.3, and arrow 3.4) to create the lowering animation. At the end of this animation, the collapse button’s icon is set to the image of the up arrow and the title of the collapse button is set to “Show Details”, which corresponds to the srcl and title1 nodes in the DMG respectively in Figure 4.15 (c).

**The Fourth Step**

Finally, the collapse button is clicked again and the info pane is raised again. By examining the newly generated DMG, we can see that this raising animation does not create any extra nodes in the DMG, as shown in Figure 4.16 (C). It only connects (arrow 4.1) the last node that has been created in the previous step with the first node that has been created in the first step, which creates a complete DMG. The reason is that this raising animation actually executes the context nodes that have been created in the first step again. After the DMG is completely created, it will not evolve, regardless of how many times the collapse button is clicked again, which provides a stable DMG for developers.

By understanding the information pane and collapse button behaviors from the browser view, and examining the topology of the flow relationships between the DMG nodes, developers can plainly
determine that \texttt{top0}, \texttt{height0}, \texttt{clip0} must be responsible for part of the information pane raising effect; \texttt{src0} and \texttt{title0} are the contexts responsible for setting the image of the down arrow and the text of “Show Less Details” respectively. So then, \texttt{height1}, \texttt{top1}, and \texttt{clip1} must be responsible for the lowering effect, \texttt{src1} is responsible for changing the down arrow to up arrow, and \texttt{title1} is responsible for setting the text to “Show Details”. Now, the developer can link to the code associated with any of the DMG nodes they are interested in for performing any changes during maintenance or debugging. Next, I discuss the conclusion drawn from this case study to answer research question 2.

![Figure 4.16 [Example]: Step 4: the UI for the Catalog Browser when the Info Pane is fully expanded again (a). The executed DOM mutator contexts are not changed because this info pane has been expanded before (b). The corresponding DMG is fully created (c).](image-url)
4.4.3.3 Conclusion

A GUI developer who wants to change the maximized height of the expanded information pane has to find how the information pane is animated from the minimized state to the maximized state and where the maximized height of the information pane is specified. However, the visual behavior of the information animation and the collapse button are achieved by a set of coordinated DOM attribute mutations such as `style.clip`, `style.top`, and `style.text`. This is the coordinated behavior understanding problem. In this case, to fully understand the animation and make some changes, the developer needs an effective way to find all the DOM attribute mutations and organize them together in an appropriate temporal order. Traditionally, the developer must manually search the code and use an ad hoc approach to put all the related code together to understand the animation. Using ScriptInsight, after clicking on the collapse button, the developer is navigated to the prerecorded DMG of the animation. Next, the developer is able to use the DMG to understand the animation and then make appropriate changes for the animation. Comparing the two scenarios of using a traditional approach or ScriptInsight, I answered research question 2 by concluding that DMG provided by ScriptInsight is useful to address the coordinated behavior understanding problem.

4.4.4 Case Studies: Other Examples

Next, I describe the other four interactive widgets presented in Figure 4.7 and listed in Table 4.2. For each particular widget, I first explain the functionality of the widget and then further describe the case where front-end developers want to modify/debug the behavior of the interactive widget. Through these descriptions, I want to argue that modifying the behavior of a rich interactive widget cannot be easily done for most cases. Therefore, developers need some technical support to assist them with these modifications/debugging, for example, ScriptInsight.

4.4.4.1 Pet Thumb (Label B)

A horizontal, “thumbnail image” scrollbar is provided in JPS 2, as shown in Figure 4.7 (B) to preview each pet in some category. When the scrollbar is used, the images’ `left` properties are mutated to cause them to slide horizontally. When an image reaches the end of the viewable region, its `clip` property is mutated to mask out the portion of the image that extends beyond the viewable region.

Consider the case in which a developer wishes to increase the size of the viewable region. They would have to determine how the `clip` property was mutated over time. Using my tool, they can hyper-link to this statement in code to examine how this calculation is performed.
4.4.4.2 Scroll Button (Label C)

The scroll button as shown in Figure 4.7 (C) is the small icon on the top left and right corners of the scrollbar. When the cursor is placed on the left or right scroll button, the thumb pictures in the scrollbar will be rolled to the left or right. These scroll icons are manipulated by DOM image elements. When the scrollbar reaches the end of the left side or right side, the courier of the corresponding image is toggled to reflect that the scrollbar cannot continue to scroll.

Suppose a developer finds that the left scroll icon does not hide when the scrollbar rolls to the end of the left side. The developer could use ScriptInsight to click on the icon and then look for a mutation of the visibility property. If they found the visibility is simply never mutated at all, they could then locate the code where the thumbnails were repositioned, to insert the visibility mutation at that location, because this code would determine if the thumbnails had finished scrolling.

4.4.4.3 Collapse Button (Label D)

The collapse icon as shown in Figure 4.7 (D) controls the sliding up and down of the information pane (E). It is an image element in the DOM. There are two places in JavaScript that set the src property. Upon examination I could tell that the reason is that the collapse icon will be changed to “up-to-down.gif” when the information pane is opened and “down-to-up.gif” arrow when the information pane is closed. Furthermore, this collapse button has a title property that displays different texts for different collapse icons when a user hovers the mouse cursor on this button.

Suppose a developer wants to change the two collapse icons to other icons or HTML elements. They have to identify the pattern of toggling the two collapse icons and then mentally map this pattern back to the JavaScript implementation. In this case, my tool can help them quickly abstract this pattern and then present it in a visual format. So, the developer can use this visual presentation to quickly locate the places where the two icons are set and then change to other icons or HTML elements.

4.4.4.4 Pet View (Label F)

When a user clicks on a pet’s thumbnail image in the scrollbar, the corresponding large image of that pet is presented as shown in Figure 4.7 (F). JPS 2 uses an opacity fading effect to make one image fade in while another image fades out. Furthermore, when a new picture is completely loaded, the previous picture’s courier is set to false. This effect is similar to the accordion row in technical detail. Here, one picture is faded in and another picture is faded out, based on the calculation of the opacity value for the two pictures, which is similar to the calculation of the height value for the expanding

107
and deflating of the accordion row. Different from the accordion row is the fact that there are two different locations for the setting of the opacity: one is used when the pictures have already been buffered by the browser, and another is the place where the pictures have to be downloaded from the server by using an Ajax request.

Suppose a front-end developer wants to change the speed of the fading effect, i.e., to slow down the speed of fading in and fading out. This code modification task is similar to the code modification task that has been given in Section 4.4.2. Therefore, the corresponding DMG provides a similar usage for this code modification task.

4.5 JavaScript Metrics

In order to better understand if my approach is truly motivated by the complexity of today’s JavaScript implementations for several existing Web applications, I have evaluated five test cases to address the following two research questions:

**RQ 3:** Is today’s JavaScript implementations of existing Web applications, such as eBay or Facebook, complicated enough to motivate my research?

**RQ 4:** What are the performance implications of using ScriptInsight to deal with existing Web applications, such as eBay or Facebook?

To answer the above research questions, I gathered metrics from JPS 2 and several popular Web sites. These measurements were taken using Mozilla Firefox 3.0.3 for Microsoft Windows.

**Table 4.3:** (# of Files) lists the number of JavaScript files downloaded for each page and (Total Lines) is the sum of their file line counts.

<table>
<thead>
<tr>
<th>Web Page</th>
<th># of Files</th>
<th>Total Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPS 2</td>
<td>3</td>
<td>1,232</td>
</tr>
<tr>
<td>eBay</td>
<td>4</td>
<td>19,682</td>
</tr>
<tr>
<td>Facebook</td>
<td>7</td>
<td>37,310</td>
</tr>
<tr>
<td>Yahoo</td>
<td>1</td>
<td>10,218</td>
</tr>
<tr>
<td>Amazon</td>
<td>4</td>
<td>5,903</td>
</tr>
<tr>
<td>Priceline</td>
<td>9</td>
<td>11,667</td>
</tr>
</tbody>
</table>
Here I first give a brief description of the pages they were drawn from. For JPS 2 I use the Catalog Browser, which has been described in detail. The eBay page is a simple list of results for searching auctions related to “iPods”. The Facebook page is the default “Profile” page for a new Facebook user. For Yahoo, Amazon, and Priceline, I used the default homepages.

Table 4.3 shows three columns of metrics for each page. The second column, number of files, counts the JavaScript files that were referenced by the page. The total lines, column three, is the sum of the files sizes (in terms of lines) for those files. In Section 4.5.1 and Section 4.5.2, I discuss some experiments and conclusions drawn from the experiments to answer the above two research questions respectively.

### 4.5.1 Complexity of Existing Ajax Applications

Table 4.4: (Average Context) lists the average number of distinct contexts that mutator statements were executed in. (Standard Deviation) lists the corresponding standard deviation. (Total Mutator) lists the total number of DOM mutator statements executed for the page.

<table>
<thead>
<tr>
<th>Web Page</th>
<th>Average Context</th>
<th>Standard Deviation</th>
<th>Total Mutators</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPS 2</td>
<td>2.2</td>
<td>1.6</td>
<td>118</td>
</tr>
<tr>
<td>eBay</td>
<td>1.5</td>
<td>0.84</td>
<td>43</td>
</tr>
<tr>
<td>Facebook</td>
<td>1.7</td>
<td>1.3</td>
<td>485</td>
</tr>
<tr>
<td>Yahoo</td>
<td>2.3</td>
<td>1.4</td>
<td>164</td>
</tr>
<tr>
<td>Amazon</td>
<td>2.0</td>
<td>0.95</td>
<td>91</td>
</tr>
<tr>
<td>Priceline</td>
<td>3.5</td>
<td>1.8</td>
<td>73</td>
</tr>
</tbody>
</table>

In this subsection, I first describe the metrics that are used. Next, I describe my methodology for collecting the metrics. Finally, I discuss the conclusion drawn from the metrics to answer research question 3.

Table 4.4 describes information about the DOM mutator statements that were executed in the page. The column labeled Average Context lists the average number of distinct calling contexts in which a statement executed. For example, considering JPS 2, each assignment statement to a DOM attribute was executed in 2.2 different calling contexts on average. The third column shows the
standard deviation. The last column lists the total number of DOM mutator statements executed for the page.

I took the metrics by triggering a measurement function injected into the code. Since these metrics measure properties of the JavaScript execution, I had to exercise the UI of the page before taking measurements. I did this by simply manually manipulating any part of the UI that did not cause the page to be changed (hence losing the script state for the page).

By looking at the results in Table 4.4, we see for which pages the calling context could be useful. Here we see that these pages either: frequently execute mutators in more than one context and/or execute some mutators in many different contexts.

In general, we see that it was common for a mutator of a DOM node to be used in more than one context. At first this could seem unintuitive because even most interactive Web pages tend to have a large amount of static content. However, this makes sense since my evaluation is only including mutations made in the JavaScript code and not any HTML attributes that are set in the static HTML or HTML generated by the server. If some attribute was going to be set only one time and never mutated, it would make sense for the developer to choose to generate the value on the server. Thus for JavaScript execution, the reuse of code from different contexts appears to be prominent for these pages.

Developers working on a particular Web page without the help of a model will have to create a mental map that connects an element of the Web page to a particular location in code. This would currently be done in an ad hoc fashion. Two possible examples are as follows.

First, a developer could scan the code to identify relevant code. From the # of files and total LOC in Table 4.3, I believe that this approach is not scalable. There is simply too much code to consider across the files. In some cases, the actual number of files delivered to the Firefox browser might be different from the number of files used by Web site developers during their development process. Still, if the code was separated into more or fewer files, this would not solve any problems related to the dynamic calling context of statements.

Second, a developer could associate an identifier such as a JavaScript function name or file with each element of the Web page. For example, they might use a particular file for all “information pane” functions. In this way, when they want to work on some code related to a particular element, they could use a text-based search to find the relevant code. However this one-to-one mapping does not appear scalable in light of the metrics from Table 4.4, because a distinct page element may be associated with code reused by several elements or for different purposes, which creates the complexity for today’s JavaScript code debugging and maintaining. This real life complexity that exists provides the answer to my research question 3.
4.5.2 Performance and Scalability

Since my tool collects a history trace of DOM mutations, I wanted to determine how much memory overhead was used for the example Web pages in Table 4.3. These measurements are listed in Table 4.5. The second column lists the memory usage of Firefox with a page loaded, after having its UI exercised, without my tool in use and the third column lists the results with the use of my tool. Here we see that the amount of memory used was never more than 4MB, which answered my research question 4.

Table 4.5: (Memory Without ScriptInsight) lists the memory usage of Firefox with a page loaded, after having its UI exercised, without my tool in use and (Memory With ScriptInsight) lists the results with the use of my tool.

<table>
<thead>
<tr>
<th>Web Page</th>
<th>Memory Without ScriptInsight(MB)</th>
<th>Memory with ScriptInsight(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPS 2</td>
<td>36</td>
<td>38</td>
</tr>
<tr>
<td>eBay</td>
<td>40</td>
<td>44</td>
</tr>
<tr>
<td>Facebook</td>
<td>68</td>
<td>72</td>
</tr>
<tr>
<td>Yahoo</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>Amazon</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>Priceline</td>
<td>38</td>
<td>40</td>
</tr>
</tbody>
</table>

Certainly the memory used will depend on the code for the page itself. For example, in the motivating example, the history for expanding and deflating one accordion bar, one time, required many trace entries. If JPS 2 was programmed differently, this number could certainly increase, but I believe that JPS 2 and the example pages in Table 4.3 are a fair representation of UI programming practices for many of today’s Web applications.

I have used my tool extensively in the exploration of JPS 2 and as part of collecting the measurements for Table 4.4. Using ScriptInsight, I did not notice any perceptible slow-down caused by the runtime tracing while interacting with the page.

Another related consideration is in the correlation of state across different DOM nodes. My current tracing mechanism keeps track of separate DOM node properties in isolation, not the state of entire DOM document. For example, a developer might want to know what the src property of an image was at the precise moment when the height of some div element reached its maximum. This tracking
is possible but would require some tradeoffs for memory consumption. In my research, I consider this part as my future work.

4.6 Discussion

4.6.1 Application

ScriptInsight has several applications for improving view-based maintenance of traditional WYSIWYG user interface design tools for Web applications. Some well-known HTML editors, such as FrontPage, are only useful for developers to maintain hyper-link styled Web user interfaces. These kinds of user interfaces use static HTML to display information and HTML is a markup language that does not provide any dynamic computing features. In this case, identifying the mapping relationship between elements on a Web page and the actual HTML code is quite straightforward.

With the new trend of Ajax applications, user interfaces of Web applications are becoming more and more similar with desktop application user interfaces. For example, users can drag and drop icons on a Web page; windows can be opened and closed with some animations. All those rich interactive and active features of Ajax applications are derived from an important technique, dynamic HTML, which provides richness and responsiveness of user interfaces of Ajax applications. Ajax applications bring some wonderful user experiences, but create a burden for GUI developers who need to understand and maintain the user interface code. The reason is that the newly introduced dynamic features cause the mapping relationship between a user interface and the actual implementation to become unclear. My tool, ScriptInsight, has been built to assist developers in this matter.

4.6.2 Manual Effort and Automation

For a GUI developer who wants to use ScriptInsight to understand the GUI behaviour implementation from a dynamic view, they will need to manually set up the environment for ScriptInsight. ScriptInsight automatically generates corresponding context and execution order information for the GUI behaviour. Next, I discuss these manual effort and automation in details.

Manual Effort

- The developer first needs to manually set up the HTTP proxy that provides JavaScript code instrumentation.
• The developer needs to point the browser to the proxy.
• The developer needs to select a widget of interest in the browser view.
• The developer needs to interact with the widget to trigger the corresponding animation(s).
• When the widget has more than one associated event, the developer has to manually select which event they are interested in.

Automation
• Execution order and context information are generated automatically based on the developer’s interaction.

From the above description of the manual effort and automation, we can see that all the procedures of the manual effort are very straightforward. However a developer normally needs to exert some effort as I discussed in the two case studies to understand the context and execution order information. ScriptInsight provides that information automatically and visualize that information to developers when they need.

4.6.3 Scope

In this subsection, I describe the kinds of programming tasks that my tool can be used for. In traditional Web applications, when an HTML Web page is delivered to a client’s browser, this Web page cannot be changed anymore hence it is static and lacks the richness. In a rich interactive Web application, user interfaces can still be mutated inside a user’s browser based on different conditions and user actions to create different UI behaviors on browser views.

Different techniques have been proposed to build rich interactive user interfaces for Web applications. My research focus is on Ajax, a popular technique for building rich interactive Web application. Many famous rich interactive Web sites such as Google map, Gmail and Facebook use Ajax extensively.

In Ajax applications, when a Web page is loaded into a browser, the HTML is parsed into a hierarchical data-structure called DOM. Standards compliant browsers such as Firefox and Internet Explorer use a common interface for the model as specified by the W3C.

While a page is active in the browser, this model is made available to any JavaScript running in the page to be queried or mutated. Any mutations made to the DOM are immediately reflected in the visual output. In this way, JavaScript can be used to achieve advanced visual effects such as panning, zooming, or other dynamic animation. My monitoring approach and tool addresses this specific
aspect of JavaScript programming.

### 4.6.4 Limitations

Here I discuss several limitations of my tool. One limitation is that my tool is only able to capture, and focus on, a standard set of HTML attributes and DOM operations. In most standards-compliant Web applications, the implementation statements that can cause visual changes to the UI are limited to these HTML attributes and DOM operations. However, if implementation code was non-standardized or able to directly draw to the browser window using pixel-level operations, for example, canvas element, a behavior-code mapping would be much more difficult or even impossible to create.

Another limitation is that developers need to trigger the UI animation of interest to obtain the corresponding DMG. However, there are some situations in which GUI developers are interested in a very specific UI animation, which is very difficult or even impossible to obtain through a developer’s guided analysis. For dealing with this problem, Mesbah’s research [68] can be used to obtain coverage of Ajax applications.

The last limitation I discuss is that the context-sensitive approach ScriptInsight used can only capture different calling context information for a DOM mutator, which is not sufficient in some situations. For example, GUI developers could choose to create different UI behaviors by using the same calling context but inputting different arguments to control how the function works. ScriptInsight cannot capture this “semantic-sensitive” calling information, e.g., the actual values of arguments, which is a limitation of my approach. For the accordion bar example, the original UI developers of JPS 2 create the two different UI behaviors (expanding and deflating of an accordion row) by calling `setHeight` function in different calling contexts. ScriptInsight uses a context-sensitive approach to capture not only the execution of the mutator that sets the height of the accordion row but also the two calling contexts, which can assist GUI developers in eliminating the ambiguity of the purpose of multiple executions of the same mutator. However, the two UI behaviors of the accordion row could be implemented by using the same calling context but with different string arguments, such as “expand” and “deflate” to control the expanding animation and the deflating animation respectively. In this case, ScriptInsight cannot distinguish the execution of the two “semantic-sensitive” contexts.

---

20 `<canvas>` is an HTML element that can be used to draw graphics by using JavaScript.
4.6.5 Threats to Validity

This subsection first describes different threats to the validity of my case studies and JavaScript metrics and then explains how my experiments can be reproduced. In regards to external validity for the case studies, I only use two typical scenarios to demonstrate the application of ScriptInsight in addressing the context-sensitivity problem and coordinated behavior understanding problem. In some situations, it is possible that DOM mutator context and DMG that are provided by ScriptInsight are not useful for some UI developers because they might want to use a different approach to visualize the context and execution order information. Therefore, a user study would be useful to evaluate if the DMG is helpful for UI developers, which will be considered as my future work.

Another issue concerning the external validity of my JavaScript metrics is that I only exercised the parts of the UI that were obvious to me, it is possible I missed some button, menu, or other widget that was not clearly marked. Since the amount of memory used was small relative to that in today’s desktop machines, I did not consider this to be a major concern.

With respect to internal validity, ScriptInsight is only able to deal with standard DOM methods such as createElement and appendChild. However, UI developers may use innerHTML statements to quickly construct HTML elements. Compared to the structured DOM tree model, innerHTML approach is unstructured and error prone. ScriptInsight does not support innerHTML statements and is not able to process these statements. After manually examining the JavaScript code of the five test cases by searching the keyword “innerHTML”, I did not find that this issue affects the validity of my results.

Finally, I describe the reproducibility of my experiments. ScriptInsight and JPS 2 are all open sources, therefore my case studies can be completely reproduced. To reproduce the JavaScript metrics for the default homepages of Yahoo, Amazon, and Priceline, an Internet archive website [46] needs to be used to retrieve the cached web pages of that particulate date when my evaluation was done. The JavaScript metrics cannot be reproduced for Facebook and eBay pages because their web pages cannot be retrieved from the archive website. In this case, the experiments can only be performed based on the current web pages of Facebook and eBay. Their web pages also contain a large amount of JavaScript code, which could be used to evaluate research question 3 and 4 in Section 4.5.

---

21 innerHTML is a property that is associated with a DOM node and it allows a developer to retrieve or set content on the node.

22 ScriptInsight has been integrated into the Mozilla Firefox Web browser and the Firebug development tool by Steven Gao, a M.Sc. student who worked with me. This new Firebug extension is called FireInsight. It integrates all the functionalities that I discussed in Chapter 4, such as DOM mutation context and DMG, into Firebug and provides a better development experience for Web developers. The reason is that Web developers currently already use Firebug intensively. FireInsight can be downloaded from https://github.com/sxgao3001/fireinsight.
Chapter 5

Related Work

In Chapter 3 and Chapter 4, I used dynamic monitoring of the execution of GUI applications to improve the understanding of existing GUI designs. One particular difficulty for GUI development and maintenance is that a large amount of GUI code can be scattered across many different source modules. Traditional WYSIWYG-based GUI editors for GUI applications do not resolve this difficulty mainly because: 1) these editors have limited ability to statically reconstruct existing GUI code, and 2) these editors do not provide an abstract model to help developers in understanding the implementation of UI behaviors.

My dynamic approaches allowed the UI implementation of GUI programs to be understood by developers in two aspects: 1) a complete design view that provides view-based navigation and view-based editing for non-animated UIs, and 2) a custom control-flow graph called DMG that provides behavior-based navigation for animated UIs.

In this chapter, I first explain related work and then present a comparison to illustrate their differences from my research. Through this comparison, I hope to show the contributions of my research. Currently, desktop and Web applications are two major types of GUI applications, and thus related work will mainly cover these two domains.

5.1 Model-View-Controller

In order to make GUI code easier for developers to understand and maintain, the Model-View-Controller (MVC) (Figure 5.1) architecture is widely used for developing GUI applications. MVC is
an architectural style in software engineering. Many popular software frameworks, for example, the Java widget toolkit Swing [89] and the Web application framework Struts [6], are typical applications of the MVC architecture.

The MVC architecture is normally used to isolate the UI code from the implementation of business logic, resulting in an application on which it is effortless to modify either the visual-appearance of the application or the underlying business rules without affecting the other. In MVC, the Model represents the data of the application; the View represents the user interfaces such as the windows, buttons, and other visual artifacts; and the Controller handles the communication between the Model and the View.

![Figure 5.1: The MVC architecture. Each box represents a component of MVC, for example, Model, Controller, or View.](image)

The background information regarding the function of the MVC architecture can be best demonstrated by using an example that illustrates the control flow of an application. First, a user interacts with the View of an application in some way, for example, clicking a button; second, the Controller processes the user’s action; third, the Controller updates the Model to reflect the user’s action; and fourth, the View uses the Model indirectly to generate an appropriate appearance.

Keeping View-code, Model-code, and Controller-code separate from one another is important for the implementation of the MVC architecture. This separation can help GUI developers in understanding/maintaining GUI implementation by only focusing on the View part. So, theoretically, GUI developers only need to focus on View-code rather than looking through Model-code and Controller-code. However, such separation is often violated because of two difficulties.

The first difficulty is investigated by Parr [79]. According to his research, programmers prefer not to enforce separation because of “fearing the loss of power resulting in a crucial UI that they cannot
generate while satisfying the MVC principle.” Instead, “they encourage rather than enforce the principle, leaving themselves a gaping ‘backdoor’ to avoid insufficient UI generation power. Unfortunately, under deadline pressure, programmers will use this backdoor routinely as an expedient if it is available to them, thus, entangling logic and display.” In many existing GUI applications that are developed based on the MVC pattern, the View implementation has to be tightly coupled with the Controller/Model, or developers do not follow the MVC architecture properly because of deadline pressures to have a finished product.

The second difficulty is illustrated by Fowler [34]. For the past decades, IDEs have been successfully used for developing information systems. IDEs are designed for “placing user interface on to databases.” The key to success of IDEs is the data aware widgets; for example, a pop-up menu is combined with an SQL query, which allows developers to quickly build a user interface that operates on a database. However, IDEs do not provide a clear boundary for separating the View part and the Model part. In this case, once logic gets complicated, it becomes hard to see how to separate the implementation of the View from the Model and the Controller in MVC. There are some other difficulties that prevent this separation; for example, some developers have insufficient training and could mix the View-code with the Model-code and the Controller-code during the development because they are not familiar with MVC.

In the above two paragraphs, I listed two scenarios that prevent a complete separation between the View and other business logic. Therefore, using the MVC architecture for GUI applications does not help in completely separating the GUI implementation from other program logic. In this case, developers still have to dig through different software modules to understand/maintain the large amount of GUI implementation.

Existing research such as Parr’s [79] addresses this problem by enforcing strict Model-View separation in HTML template engines. In [79], Parr gives a formal definition of the Model-View separation and proposes a new template engine, which strictly enforces a list of rules of separation by design. His research attempts to achieve a complete separation for each of the Model, View, and Controller. However, even the View itself still contains a large amount of code because the graphical user interfaces can constitute as much as half of the source code in typical projects [75]. In this case, my research is different from his research because I focus on how to assist GUI developers in understanding the implementation of the user interfaces by using dynamic views as entry points. Thus, developers do not have to make the effort to enforce strict separation during the design phase or modify their original designs to enforce separation for better code maintenance.

---

23 A template engine is an application that processes Web templates and data from a database to generate Web pages.
5.2 Program Comprehension

Some researchers [12][20][22][25][42][80][99] study the reverse-engineering of source code to recover high-level models such as architectures from existing applications. Their aim is program comprehension and documentation. These projects are related to my work because my research also does reverse-engineering to recover models from existing GUI applications. Developers can use these models as assistance to explore the existing GUI implementation. My research extracts the WHT (Widget Hierarchy Tree) model of a user interface and the DMG (DOM Mutation Graph) model of a UI behavior from an existing GUI application. In the remaining part of this section, I discuss the related research from three different aspects. First, I discuss some model-based reverse-engineering research that is used to assist developers in understanding existing systems. Second, I discuss the related research based on how they extract models: the static approach or the dynamic approach. Third, I discuss the related work that is used to present the recovered models to developers to assist in program comprehension.

5.2.1 Model-Based Approaches

In [42], Hassan and Holt describe semi-automated tools that parse the source code and binaries of Web applications and extract relationships among different components of the applications. The models that their tools create can assist Web developers in understanding the high-level architecture of traditional HTML and template-based Web applications. In their research, each component represents a functionality of a Web application, for example, the checkout functionality for an online selling system. The extracted components and their relationships are visualized through a traditional box-arrow diagram, which is used to create an architecture diagram. Their research assists developers in understanding the high-level architecture of Web applications.

My WHT model and DMG model are more fine-grained than Hassan and Holt’s. In [42], their model is only used to navigate developers to different software modules, which offers limited assistance for developers in code understanding and modification. The reason is that their model is coarse-grained and can only navigate developers to a function or a class. In my research, the WHT model and DMG model are fine-grained enough to help developers in exploring the implementation-level details; for example, developers can be navigated to the implementation-level statement for a widget or a UI state transition. Furthermore, my research deals with GUIs and not whole applications.

The BCM (Behavioral Concern Modeling) approach and a supporting tool are presented by Lai and Murphy [54]. Using this approach, a developer first iteratively builds a FSM (Finite State
Machine) behavioral model based on their concern and then maps different pieces of the model to the source. This procedure is performed through two interchange activities. First, a developer looks for code related to the concern and puts the states and transitions in the model based on their understanding of that code. This activity is manually performed. Second, a developer uses their understanding of how the concern behaved to extend the model by adding new states and transitions. Instead of manually extending the model by tracing a control flow or data flow in the code, Lai and Murphy use a relatedness query supported by the BCM tool. After the model is built, the developer can use this model to bridge the gap between their concern and the real implementation. Their model is created semi-automatically. In my research, the WHT and DMG models are created automatically without users’ interactions, because my research only deals with the GUI concern.

5.2.2 Static Approach vs. Dynamic Approach

Two important methodologies that are used to recover models in reverse-engineering are static analysis and dynamic analysis, for example [84] and [5]. Static and dynamic analyses provide different tradeoffs: static analysis can explore many of the potential program paths in a program and dynamic analysis is precise for a particular program execution.

The research that uses static analysis, such as [99][4][72], does not execute applications. Instead, it only analyzes the source files, binary code, and any other static information. By using this analysis, the components and the static relationships with each other in an application can be revealed. The results of the static analysis method allow a first approximation of the structural views of an application [23]. Staiger et al. [84] use static whole-program analysis for reverse-engineering GUIs. Through their tool, the hierarchy structure of a program’s user interface, the attributes of each widget, and the corresponding values can be extracted. Furthermore, the extracted information also includes the GUI events that could occur at runtime and the corresponding event handlers. This information can provide a useful high-level guideline for GUI developers to understand existing user interface implementation. However, because of some well-known restrictions of the static program analysis [92], their approach cannot extract all the user interface information. For example, if some parts of a user interface are generated based on a loop and the loop’s boundary condition can only be obtained at runtime, then the static analysis cannot provide any useful information for these specific user interface parts. In the worst case, if many parts of a user interface are generated in a dynamic way, their research can only provide very limited assistance for developers. Furthermore, the recovered results of their research have not been used as input to a WYSIWYG-based tool and they do not describe how the statically recovered model would be mapped back to source code changes.
Dynamic analysis requires developers to execute their applications [21]. Next, they ensure components of interest to them are exercised during execution. During the actual execution of an application, all the states that have been recorded by the analysis are limited to a particular time and some specific user action, i.e., the information is a “snapshot” of the current running application. This is similar to memory dump (or core dump) [66], which serves as a useful debugging tool in operating systems. For example, Antoniol [5] used dynamic analysis to derive a precise model for Web applications. However, their analysis alone focuses on recovering the high-level model of an application structure instead of recovering the model for graphical user interfaces or UI behaviors. In this thesis, I have researched an approach in which one particular dynamic view or UI behavior can be selected under the developer’s control. Based on the selection, my tools used the dynamic approach to create WHT and DMG models.

5.2.3 Presenting Recovered Models to Developers

Recovered models can only become useful if they can be presented properly to developers. Because recovered models normally contain a large amount of recovered information, efficient visualizations are required. There are three main categories of existing visualization approaches.

First, UML (Unified Modeling Language) as a standard modeling language is widely used to present recovered information to developers. In [14], Chung and Lee proposed using component diagrams of UML to model a Web application. Furthermore, the navigations among the components are described by using dependencies among components in the component diagrams [14]. Conallen discusses an approach for modeling Web-specific elements by using UML [17]. Through the UML extension mechanism, Conallen showed class diagrams of specific Web elements. In addition to the class diagrams, Bellettini et al. [10] propose WebUml to generate state diagrams for Web applications.

Second, custom graphs are often used to present the recovered models for developers. For example, in [26], a sequence of web pages in a Web application are accessed by a user along their working sessions. Next, the complete set of the execution traces can be represented by a directed graph, called TG (Transition Graph). Each node in a TG represents a web page, and a directed edge between two pages represents a transition from the starting page to the target page. However, as pointed out in [24], even if the considered application is small/medium sized, TGs are very complex. So, clustering rules are applied to TGs, and each group of clustered nodes will be replaced by a single node representing that cluster. This will lead to generating smaller-size TGs, called CTG (Clustered Transition Graph).
Third, if the recovered information is extremely large, developers in this situation are more interested in the high-level topological characteristics of the information instead of the details of the recovered information. For example, WebAware [83] presents the reversed Web application information as a solar system metaphor based on a visualization technique, CyberGeo Maps [43].

In Chapter 3, WHT models are presented to developers by using rendered user interfaces. This model visualizing approach is different from any approaches that have been discussed in the above three categories. My approach provides intuitive understanding of the models by using the actual user interfaces. Developers now understand the models in a WYSIWYG way.

Using actual user interfaces to guide developers in understanding models and explore existing code has been studied in [69]. In his research, Michail introduced a tool to provide GUI-guided browsing of source code. His objective was to allow developers to find where in the code a feature was implemented, based on how code was related to the GUI. For example, to find “spell checking” code, users could locate the code that executed when the spell-checking menu was selected. Similar to my approach, he uses a GUI as an entry point into the lower-level implementation details. However, my research is different from the research in [69], because my research is concerned with providing a mapping for programmers to the implementation-level details of GUI implementation itself rather than program logic related code such as spell-checking. Furthermore, compared to my research, his research does not deal with the use of a WYSIWYG methodology and does not consider editing of code.

In Chapter 4, a DMG provides visual presentation for the recovered control flow information. My research uses custom graphs to visualize the recovered models. Similar to CTG, DMG also attempts to simplify the recovered model information. In my research, I used a trace compression approach to convert a complete trace to an abstract trace. GUI developers who work on this abstract trace would obtain understanding for the implementation level details of a particular UI behavior, which is different from the research in [24].

5.3 GUI Development

Creating good graphical user interfaces requires considering the appearance, careful design for usability, and programming for the concrete implementation. My proposed contributions are to make improvements at the implementation level, so I limit my discussion of related work to projects with similar goals.
Myers and Rosson reported results of a survey of user interface programming in 1992 [76]. Based on their investigation on 74 responses, "an average of 48% of the code is devoted to the user interface portion. The average time spent on the user interface portion is 45% during the design phase, 50% during the implementation phase, and 37% during the maintenance phase". After 20 years, user interfaces become much more sophisticated comparing to the user interfaces in 1992. However, there are no much improvements for the tools that are used to build the user interfaces. In my research, I addressed this problem.

In the Web UI domain, most existing work on modeling UI-intensive Web applications focuses on development but not specifically maintenance and debugging. For example, Yu et al. [105] introduce a framework for the integration of presentation components in mashup applications. Trigueros et al. present a model-driven approach, the RUX-Model, for the design of rich Internet applications [97]. Valderas et al. introduce an approach to support the coordinated work between Web UI designers and analysts during the development of a Web application [98]. In [82], Rossi et al. use a model-driven approach to transform conventional Web applications into rich Internet applications by applying refactoring at the model level. Meliá et al. propose a model-driven development methodology that extends a traditional Web-modeling methodology for use with the Google Web Toolkit [62].

Using finite state machine models to present GUI behaviors has been studied in [7]. Their research describes a Java toolkit called SwingStates that can assist non-expert developers in the development of GUIs. The novel part of their research is that they use finite-state machines to describe the behavior of interactive UI systems. However, their research is concerned with how to create a user interface instead of reversing-engineering from an existing user interface.

5.4 GUI Testing

There are researchers who work on the GUI testing domain [63]. For example, in [64] and [104], EFG (Event Flow Graphs) and EIG (Event Interaction Graphs) have been introduced, respectively. The nodes in these graphs represent events; an edge represents the relationship between a pair of events. There is an important difference between my models and theirs: each node in a WHT model is a widget, and each node in a DMG model is a DOM mutator context, but their models consider each node as an event. So, their models are more useful in assisting developers in generating test cases automatically, since their models reveal the potential sequence of different events. However, my models are better in assisting developers in exploring statement-level GUI implementation.
Mesbah et al. introduced a tool called CrawlJax in [68]. Their research uses a dynamic approach to crawl Ajax-based applications by triggering the event handlers in the code. After crawling, a state-flow graph is constructed. In this graph, each node represents the snapshot of the DOM tree for a Web UI after some event handler is triggered; each edge in the state-flow graph represents the clickable element that transforms one state to another state. This state-flow graph can be used to provide automated testing of Ajax applications. Different from my research, their research has been used for testing of Ajax applications but not specifically for interactive debugging.

Memon et al. discuss a dynamic process in which the software’s GUI is automatically “traversed” by opening all its windows and reverse-engineering all widgets. This information can be used as feedback for automatically generated GUI test cases [65]. Compared to my research in Chapter 3, their research is not concerned with view-based maintenance of GUIs. In general, maintenance tasks could require both human guided code inspection and automated test cases. View-based navigation addresses the first, whereas their research addresses the second. Their research also does not address tool support for improving WYSIWYG view-based editing.

In [61], McMaster et al. present how to use calling context information collected during a GUI program’s execution to solve the GUI test suite reduction problem (i.e., finding a minimal satisfactory test set). Their research considers two GUI test cases to be equivalent if they generate the same set of call stacks after execution. This new call-stack coverage criterion can be used to address the challenges for testing GUI-intensive applications, which are difficult to handle by some other criteria such as statement or branch coverage. Similar to their research, I also use a calling context to distinguish two artifacts such as the expanding and deflating behaviors of an accordion row, in Chapter 4. However, calling contexts are used in my research to resolve the ambiguity of the different UI behaviors instead of GUI test cases.

5.5 Evolving Legacy GUI Systems

Several papers address reverse-engineering of systems in order to evolve legacy systems to more modern user interfaces. In [67], Merlo et al. discuss a method for reverse-engineering user interfaces to turn console-based text interfaces into GUIs. Bodhuin et al. [11] have a similar approach for migrating systems to the Web. In contrast, my research has not investigated how to port an interface to another platform. My research has focused on how to assist developers in understanding and making changes to existing user interfaces.
5.6 Tool Support

5.6.1 JavaScript Monitoring Platforms and Programming Tools

Recently, research on Ajax has caused growing interest in JavaScript monitoring platforms. AjaxScope [51] can perform on-the-fly parsing and instrumentation of JavaScript code. In the research, they use a proxy server to rewrite the JavaScript code by injecting instrumentation code. Similar to my approach, when the JavaScript code is delivered to the client side browsers, the injected JavaScript can be used to monitor the behavior of Web applications. However, their research considers the correctness and performance of the JavaScript code and does not address the development and the maintenance of user interfaces.

Due to the popularity of Ajax-based applications, there is an increasing demand for JavaScript programming tools. One representative tool for developing Ajax applications is the Firebug [32] extension for the Mozilla Firefox browser. Using Firebug, a developer can simply click on a rendered element in the browser and be hyper-linked to an expanded tree-view of the corresponding DOM element. A developer can inspect the low-level attributes of that specific DOM object and understand its context relative to its ancestor and children objects. Although this practice is useful, Firebug still does not provide any help for the developer in understanding the connection between a DOM node and the JavaScript that acts on the DOM. Essentially, my research in Chapter 4 addresses this mapping between DOM and JavaScript.

5.6.2 WYSIWYG GUI Editors for Rich Internet Web Applications

Dreamweaver is a representative GUI editor that has been widely used for developing user interfaces of Web applications. Dreamweaver provides four important views for GUI developers: design view, HTML view, live view and DOM view. For a rich interactive web page, the design view is just an estimation of how the page could look at runtime. The reason is that Dreamweaver uses a static approach to recover the web page, and a rich interactive web page normally contains many dynamic UI features that cannot be rendered properly through a static code analysis approach. For web pages that contain many dynamic features, design views are very different from runtime appearance in many of the cases. This design view is similar to the static design view, which has been described in Chapter 3, and can only provide very limited assistance for developers. In Dreamweaver, a design view corresponds to a HTML view. So, when a GUI developer clicks a widget on a design view, the
corresponding HTML element of that widget will be highlighted in the HTML view (view-based navigation). Furthermore, when a widget is changed in the design view, the corresponding change will be reflected in the HTML view (view-based editing). A design view and the corresponding HTML view do not provide any assistance for developers in understanding the implementation of a UI behavior or animation, because understanding their implementation requires the dynamic execution of the code.

A live view is the runtime appearance for a web page, which is the same as the appearance that a GUI developer can see from their browser. In Dreamweaver, a live view is rendered by using the WebKit [101] rendering engine, which also renders web pages for Apple’s Safari web-browser. This live view gives GUI developers an exact preview of rich interactive web pages, which are just as a Safari browser would render them. The DOM view shows how a browser interprets with the underlying tree-structured representation of a web page, i.e., DOM. For example, when some interaction is performed on the live view by a GUI developer, the DOM view simultaneously presents how this interaction affects the web page's DOM representation.

In an Ajax-based Web application, a UI behavior varies over time, based on mutations of state made from JavaScript code. The JavaScript code mutates the DOM representation to create the UI behavior. Different from the HTML view, this DOM view does not present the actual code that generates the visual behavior in the live view. Instead, this DOM view is only a tree-structured representation of the entire web page.

The live view and DOM view provide limited assistance for developers in understanding the implementation for a UI behavior or animation. A developer can use live view to map a UI behavior to a set of DOM elements that represents the different states of the behavior. However, existing GUI editors such as Dreamweaver do not provide any assistance for developers in understanding the connection between a DOM node and the JavaScript code that acts on the DOM. Therefore, the gap between the DOM node and the actual code that mutates the DOM node still has to be bridged if a developer wants to change or understand the UI behavior. My research in Chapter 4 bridged the gap by using a DMG to provide the high-level abstraction of a UI behavior. Furthermore, this DOM view sometimes could be overwhelmed by the large number of generated DOM nodes that come from some dynamically generated HTML tags, which makes the DOM view less useful.
Chapter 6

Conclusions

6.1 Thesis Summary

In this thesis, I have discussed view-based maintenance for graphical user interfaces of desktop and Web applications. One difficulty in maintaining GUIs is that the relationship between the view and source code is not always clear. I showed that one important reason is because a large amount of user interface code is spread across the decomposition of applications. For example, in a case study I saw only 1 out of 5 dynamic views was built out of less than 10 classes. A popular approach to develop and maintain GUIs is to use “What you see is what you get” editors. For example, Swing Designer and Dreamweaver, they allow developers to work directly with design views instead of the scattered source elements. Developers who work with these design views can easily explore the implementation for each specific UI piece that they are interested in. However, traditional GUI editors have two major defects that strongly restrict their abilities.

6.1.1 Defect 1: Limited Reverse-Engineering Capabilities

The first defect is that traditional GUI editors have limited reverse-engineering capabilities. The implementation of a complete GUI design includes both static design and dynamic design. The static design and dynamic design of a user interface are often tangled together in the user interface implementation. So the appearance of some parts of a user interfaces are dynamically computed results.
Traditional GUI editors such as Swing Designer cannot properly reverse-engineer existing GUI code that contains dynamic computation code. In this case, GUI editors can only provide incomplete design views for developers to work on. Based on these incomplete design views, GUI developers will lose all the view-based navigation feature and view-based editing feature for the widgets that cannot be displayed to them. Furthermore, some widgets can be rendered by using a traditional GUI editor, but they might not be edited properly because they lack dynamic runtime context to help developers make editing decisions. For example, the navigation panel editing example has been discussed in the motivating examples of Chapter 3. In the example, the developer loses the expected WYSIWYG feature for GUI editing. Sometimes a dynamic context is useful for developers to make decisions about potential changes.

In Chapter 3, I investigated the combination of a hybrid dynamic and static approach to allow for view-based maintenance of GUIs. Dynamic analysis provides a concrete context in which maintenance can be performed, while static checking ensures that changes propagated from the view to the source are predictable. To use FreezeFrame, a GUI developer needs to first choose a user interface of interest during the execution of a GUI application. To choose this user interface, the developer first executes the application and then makes sure the dynamic view the developer is interested in is rendered during that execution. Next, the developer notifies FreezeFrame to create a “snapshot” for the dynamic view at the particular moment. This “snapshot” is a dynamic model that is created by recording all the widgets, widget properties and widget relationships on the dynamic view. In my research, I use AspectJ to collect this model. I created an aspect, GUIAspect, to monitor three important kinds of information related to the user interface, the creation of widget classes, information regarding any changes to the properties of a widget and information regarding the containment relationship between container widgets and their child widgets. Finally, the static design model that is normally used by Swing Designer is replaced with this dynamic model. Based on this dynamic model, a developer can see a complete design view and they can work on this complete design view to do view-based navigation and view-based editing. Furthermore, an editing constraint is used to prevent developers from editing any widgets that do not have a one-to-one mapping with the source statements.

To evaluate FreezeFrame, I first measured how many classes contribute implementation for generating the runtime main window of each test case. From the metric, we can see that in at least 4 out of the 5 cases over 10 classes were used. This metric supports my claim that maintaining GUIs can be difficult and that view-based navigation is useful. Second, I measured how much of some view for each program could be reverse-engineered using a traditional GUI editor. From the results, we can see that in 4 out of the 5 cases less than 50% GUI designs can be recovered and displayed. This
supports my claim that design code and program logic is often tangled, making design model recovery difficult. Third, I measured that my approach enabled at least 50% of the dynamic view to be editable in 4 out of 5 cases. I showed an addition of between 1.2x and 5.3x more design information in 4 of the 5 cases I looked at. Furthermore, the two larger applications appeared to benefit the most. This supports my claim that dynamic context provides more opportunities for design view-editing.

### 6.1.2 Defect 2: No Explicit Support for Coordinated Behaviour

The second defect of traditional GUI editors is that there is no technical support for GUI developers to understand the implementation of a UI behavior. The difficulty of understanding UI behavior implementation is that one UI behavior normally contains different UI states, and these states transition based on a user’s interactions. It is difficult for GUI developers to map observed state transitions to the actual source code that implements them.

Traditional GUI editors such as Dreamweaver do not provide explicit support for GUI developers to understand GUI behavior implementation for Ajax applications. This is because of the static nature of traditional GUI editors. However, finding a direct mapping between a GUI behavior and the corresponding implementation is not as straightforward as view-based navigation. The reason is that a GUI behavior is normally implemented by a set of DOM mutators and they are executed in an appropriate temporal order (i.e., the coordinated behavior understanding problem). Furthermore, each DOM mutator could be executed in multiple calling contexts to create completely different GUI behaviors (i.e., the context-sensitivity problem).

In Chapter 4, I have studied the problem of implementation complexity for understanding the UI behavior of a rich interactive Web UI: the context-sensitivity problem and the coordinated behavior understanding problem. I proposed an approach which leverages execution order and calling context so that developers can explore the code from the browser view. To solve the first problem, I studied the use of context-sensitivity to capture not only the execution of individual statements but also the state of the call stack, which can help distinguish among multiple executions of the same statement. To solve the second problem, I studied the use of a custom control-flow model, DMG, which developers can use to map causal relationships, which have been seen in the browser view, to script source code. The DMG model was introduced to present the obtained execution order and context information to developers for a better understanding of the behavior of the UI.
I presented some script complexity metrics for popular websites to further motivate the need for my interactive script development approach. I found that many of the sites that I measured included significant complexity based on the number of calling contexts for a given statement. To demonstrate how the DMG could help, I presented two detailed case studies from the open-source JPS 2 Ajax application. The first case study is the “Accordion Bar” example. I used this case study to evaluate the usefulness of the calling context to address the context-sensitivity problem. The second case study is about the “Info Pane” and “Collapse Button” example. I used this case study to evaluate the usefulness of the DMG model to address the coordinated behavior understanding problem.

6.2 Future Work

6.2.1 WYSIWYG Change Impact Problem

A common question to ask about my approach is, “What if a GUI developer wants to edit only a single particular widget instance without affecting other widget instances that are implemented through the same code abstraction?”. I have not solved this change impact problem in my current implementation and it would not be compatible with current GUI editor methodology. For example, the most straightforward code generation approach, used by all GUI editors that I know of, is to project changes of a widget directly to changes on the source statements which influence the widget. For example, if a widget was changed from blue to green in the editor, the editor would change some statement “w.setColor(BLUE)” to “w.setColor(GREEN)”. This may cause problems if the widget w references any other widgets during some execution of the program.

Here, I summarize two common situations when view-editing a widget in a dynamic context is useful, to make sense of this question.

First, in the case that the developer wants to edit design information and also the developer would like all widget instances that are implemented through the same code to be changed in the same way, then this is supported by my approach. In the traditional approach, there might be no widgets displayed at all.

The second case is one where the developer would like to change some widget’s design information without affecting some other widget’s design, which is implemented through the same code. In this case, all widgets currently share this design information. However, now the developer has decided to partition the set of instances created by the same code into two sets: one with the previous design and one with the new design. This requires some way to disambiguate which
instances will belong to which set. This can’t always be provided from the editor view, because editors do not support any kind of conditionals. In some (but not all) cases, these sets of widgets could be distinguished based on more sophisticated data-flow analysis. I leave this “context-sensitive” editing of widgets to future work.

6.2.2 Improving DMG

So far, ScriptInSight relates two kinds of control-flow information: calling context and the execution order of different mutator contexts, to the mutations of DOM states. Understanding this control-flow information is necessary to perform maintenance to existing JavaScript source code. Although this understanding is necessary, it is not always sufficient. Because JavaScript is based on objects, more complex data-flow relationships may need to be understood which span across disjoint calling contexts. My tool currently does not provide a way to monitor this data-flow. The fact that I do not monitor data-flow is one reason that the memory consumption is quite modest. If additional data-flow information were provided the memory consumption would need to be revisited.

The approach I have described for ScriptInsight focuses only on monitoring DOM state and not any other part of the JavaScript implementation. In Ajax applications, some parts of the script would be directly responsible for the DOM, while other code acts to process XML or JSON messages. My tool does not provide support for this lower-level message processing code since it is not directly involved in producing interactive effects or animation. In the future I hope to research if the manipulation of XML and JSON messages could be monitored using a similar approach that I had used here for the DOM.

6.2.3 Automated GUI Testing

Today, GUI implementation is becoming more and more complicated and human guided analysis is very tedious and often not sufficient. In my research, both FreezeFrame and ScriptInsight use a dynamic analysis approach. However, dynamic analysis always has the code coverage problem. In my research, I used a human guided approach to explore different user interfaces or GUI behaviors. In this case, some user interfaces or GUI behaviors could not be triggered and some UI related code could not be traversed. Automated GUI testing tools that traverse the code automatically help with the code coverage problem. In the future, I hope to use Memon’s GUI Ripper [64] and Mesbah’s CrawlJax [68] to address the code coverage problem for FreezeFrame and ScriptInsight respectively.
References


Appendix

More Examples for FreezeFrame

This appendix presents the screenshots for five open source GUI applications, CrosswordSage, Java-Chess, JMSN, GanttProject, and FreeMind. For each application, I present the screenshots of their main windows in the following three different situations:

1. The screenshot from the execution of the application.

2. The screenshot from Swing Designer.

3. The screenshot from FreezeFrame.

Furthermore, I also identify the differences between the runtime appearance of each main window and the rendered result of each main window from FreezeFrame. By looking at these differences, we can see that they are minor.

**CrosswordSage V0.3.5**

From Figure A.2, we can see that the widget of WordSolverPanel (the panel under the menu bar in Figure A.1) is missing. However, a GUI developer is still able to find the WordSolverPanel class and open it by using Swing Designer. In this case, the WordSolverPanel class can be rendered properly and the developer can edit it in a WYSIWYG way. Therefore, we say that the main window of CrosswordSage can be recovered completely by Swing Designer. However, as I discussed in Section 3.1, the developer now loses the global context of the CrosswordSage main window when they try to edit the WordSolverPanel widget. This could cause some problems as I discussed in Section 3.1.
Figure A.1: Screenshot from CrosswordSage program execution.

Figure A.2: Screenshot from Swing Designer for CrosswordSage class.
Difference

1. The icon of the main window in Figure A.1 is different with the icon in Figure A.3. This is because the developer did not set an icon for the main window (JFrame). In this case, the default icon (the coffee cup) is put on the frame by the JVM at runtime.

Java-Chess (JChess) V1.01a3

Difference

1. The icon of the main window in Figure A.4 is different with the icon in Figure A.6. This is because the developer did not set an icon for the main window (JFrame). In this case, the default icon (the coffee cup) is put on the frame by JVM at runtime.
Figure A.4: Screenshot from Java-Chess program execution.

Figure A.5: Screenshot from Swing Designer for Java-Chess class.
Figure A.6: Screenshot from FreezeFrame for Java-Chess class.

JMSN V0.9.9b2

Figure A.7: Screenshot from JMSN program execution
Figure A.8: Screenshot from Swing Designer for JMSN class.

Figure A.9: Screenshot from FreezeFrame for JMSN class.
From Figure A.8, we can see that the main window of JMSN cannot be rendered at all by Swing Designer. However, in the Section 3.6 of this thesis, I counted the size of the editor’s static model (the main window) of JMSN as “50”. This is because I counted the number of widgets and properties based on the content of the Property Editor (highlighted in Figure A.8). This is because the widgets or properties that can be displayed in the Property Editor can also be used as view-based navigation and view-based editing even though they cannot be rendered on the design view by Swing Designer.

Difference

1. The combo box in Figure A.7 is different from the combo box in Figure A.9.

When Swing Designer renders JComboBox, it puts "Combo item 1" there to help developers in understanding JComboBox.

**GanttProject V2.0.1**

![GanttProject V2.0.1](image)

*Figure A.10: Screenshot from GanttProject program execution.*
Figure A.11: Screenshot from Swing Designer for GanttProject class.

Figure A.12: Screenshot from FreezeFrame for GanttProject class.
Differences

1. There are 5 items in the table of Figure A.12 (the table under the “Gantt project” picture) and the runtime appearance of the Gantt project does not have those items in its table.

This is because Swing Designer renders JTable by offering some example items to help developers in understanding the widget.

2. Some buttons are grey in Figure A.10, but they are colorful in Figure A.12, for example, the save button.

This is because I have not yet implemented a pointcut for catching when widgets are disabled (i.e. “grayed out”) as discussed in Section 3.6.2.

3. There are two fixed-width components (marked with squares) between the buttons in Figure A.12.

These two components are invisible in Figure A.10, and Swing Designer makes them visible to help developers in editing them.

**FreeMind V0.8.0**

![FreeMind Screenshot](image)

*Figure A.13: Screenshot from FreeMind program execution.*
Figure A.14: Screenshot from Swing Designer for FreeMind class.

Figure A.15: Screenshot from FreezeFrame for FreeMind class.
Differences

1. The combo box in Figure A.13 is different from the combo box in Figure A.15.

As I had explained previously, Swing Designer renders JComboBox by offering an example item "Combo item 1".

2. Some buttons are grey in Figure A.13, but they are colorful in Figure A.15, for example, the save button.

This is because I have not yet implemented a pointcut for catching when widgets are disabled (i.e. “grayed out”) as discussed in Section 3.6.2.