# Testing and maintenance of graphical user interfaces

Valéria Lelli

Thèse

THÈSE INSA Rennes
*sous le sceau de l'Université Européenne de Bretagne*
pour obtenir le grade de
DOCTEUR DE L'INSA DE RENNES
*Spécialité : Informatique*

présentée par
Valéria Lelli Leitão Dantas

ÉCOLE DOCTORALE : MATISSE
LABORATOIRE : IRISA/INRIA

# Testing and maintenance of graphical user interfaces

**Thèse soutenue le 19 Novembre 2015**
devant le jury composé de :

**Pascale Sébillot**
Professeur, Universités IRISA/INSA de Rennes / *Présidente*

**Lydie du Bousquet**
Professeur d'informatique, Université Joseph Fourier / *Rapporteuse*

**Philippe Palanque**
Professeur d'informatique, Université Toulouse III / *Rapporteur*

**Francois-Xavier Dormoy**
Chef de produit senior, Esterel Technologies / *Examinateur*

**Benoit Baudry**
HDR, Chargé de recherche, INRIA Rennes - Bretagne Atlantique / *Directeur de thèse*

**Arnaud Blouin**
Maître de Conférences, INSA Rennes / *Co-encadrant de thèse*

# Testing and Maintenance of Graphical User Interfaces

Valéria Lelli Leitão Dantas




# Contents

# Abstract

Graphical user interfaces (GUIs) are integral parts of interactive systems that require interactions from their users to produce actions that modify the state of the system. While GUI design and qualitative assessment is handled by GUI designers, integrating GUIs into software systems remains a software engineering task. The software engineering community takes special attention to the quality and the reliability of software systems. Software testing techniques have been developed to find errors in code. Software quality criteria and measurement techniques have also been assessed to detect error-prone code.

In this thesis, we argue that the same attention has to be investigated on the quality and reliability of GUIs, from a software engineering point of view. We specifically make two contributions on this topic. First, GUIs can be affected by errors stemming from development mistakes. While the current GUI testing tools have demonstrated their ability in finding several kinds of failures (*e.g.*, crashes, regressions) on simple GUIs, various kinds of faults specific to GUIs have to be identified and studied. The first contribution of this thesis is a fault model that identifies and classifies GUI faults. This fault model has been designed empirically by performing a round trip process between the analysis of the state-of-the-art of GUI design and bug reports. For each fault we illustrate which part of a GUI is affected and how that fault occurs as GUI failure. We show that GUI faults are diverse and imply different testing techniques to be detected. We then develop GUI mutants derived from our fault model. These mutants are freely available for developers of GUI testing tools to evaluate the ability of their tool in finding GUI failures.

Second, like any code artifact, GUI code must be maintained and is prone to evolution. Design smells are bad designs in the code that may affect its quality. Maintenance operations can be performed to correct design smells. As for the first contribution, we focus on design smells that can affect GUIs specifically. We identify and characterize a new type of design smell, called *Blob listener*, which is a GUI specific instance of a more general design smell: God method, that characterizes methods that "*know too much or do too much*". It occurs when a GUI listener, that gathers events to treat and transform as commands, can produce more than one command. We propose a systematic static code analysis procedure that searches for *Blob listener* that we implement in a tool called InspectorGuidget. Experiments we conducted exhibits positive results regarding the ability of InspectorGuidget in detecting *Blob listeners*. To counteract the

use of *Blob listeners*, we propose good coding practices regarding the development of GUI listeners.

# Chapter 1

# Introduction

## 1.1 Context

Software systems usually rely on user interfaces for being controlled by their users. Such systems are then called interactive systems. User interfaces can take various forms such as command line interfaces, vocal interfaces, or graphical user interfaces (GUI). GUIs are composed of graphical interactive objects called widgets, such as buttons, menus, and text fields. Users interact with these widgets (*e.g.*, by pressing a button) to produce an action[1] that modifies the state of the system.

The development of GUIs involves multiple roles. The GUI design and qualitative assessment is handled by GUI designers. Software engineers then integrate GUIs into interactive systems and validate this integration using software testing techniques. GUI development and validation is a major task in the development of an interactive system since GUIs may achieve up to 60% of the total software [Mye95, Mem07]. Besides, a multiplication of human interface devices (HID, *e.g.*, tactile screen, gyroscope) and a diversification of the platforms (tablet, mobile, *etc.*) has been experienced over the last decade. As a result, software testers face user interfaces that have to be tested on multiple platforms with various HIDs, which increases the testing cost. Validating, from a software engineering point of view, user interfaces and in particular GUIs is now a crucial challenge. Similarly to the attention that has been paid to the validation of software systems since the last decades, we argue in this thesis that software testing must be tailored to consider GUI specific defects. Indeed, GUIs have to be considered as special software artifacts that require dedicated software testing techniques. Such techniques can rely on mainstream software testing techniques, as we detail in this document.

## 1.2 Challenges and Objectives for Testing GUIs

GUIs rely on event-driven programming: input events produce by users while interacting with a GUI are captured to be then treated by the system. Most of standard widgets

[1] Also called an event [Mem07] or a command [GHJV95, BL00].

enable user interactions composed of a single input event (*e.g.*, press). Such widgets work similarly across mainstream GUI toolkits[2] and mainly rely on the use of a mouse and a keyboard. Current GUI testing approaches have targeted GUIs composed of standard widgets (*aka.* standard GUIs [LBBC15] or WIMP GUIs [vD00])[3]. However, the evolution of GUIs and how they are developed and maintained require increasing efforts in the GUI testing domain as discussed below.

First, an ongoing trend in GUI design is the shift from designing "[user] interfaces to designing [user] interaction" [BL04]. The underlying goal is to embrace the diversity of HIDs and platforms to design user interfaces more adapted and natural to users, called post-WIMP interfaces. Post-WIMP GUIs require specific, *ad hoc* widgets [BL00, BL04, BB10] developed to fulfil a need that standard widgets cannot meet. Post-WIMP GUIs exploit the following mechanisms to be more adapted to users:

1. *Data are dynamically presented in the widget to be directly manipulated.* Post-WIMP GUIs use direct manipulation through continuous and direct interaction with objects, for instance, users interact directly with underlying data (*e.g.*, 2D or 3D objects) to produce the target action.

2. *Ad hoc interactions* are provided to interact with the data and their associated widgets. Developing and testing these interactions is more complex than interactions from standard widgets. They involve multiple events triggered (*e.g.*, voice and gesture events) by one or more modern HIDs such as tactile screen, gyroscope, or eye tracking. An example is the bimanual interaction that can be performed in a multi-touch screen for zooming shapes using two fingers as it is typically more and more the case with mobile phones and tablets.

The aforementioned specificities in GUI design demand different ways in testing a GUI. For example, testing a GUI consists of verifying whether a user interaction produces the expected action. In WIMP GUIs, the interactions are not tested since they are composed of a single event, which is embedded in a GUI toolkit. For example, the interaction "*pressing*" the left mouse button has the same behavior on all GUI platforms and thus the target is to test the action resulting after the execution of such an interaction. In post-WIMP GUIs, however, the *ad hoc* interactions involve multiple events, they have some special properties (*e.g.*, feedback) that should be tested while they are performed. Thus, we argue that a transition must occur in the GUI testing domain: move from testing WIMP GUIs to testing post-WIMP GUIs.

One major challenge with this transition is that the GUI developers and testers must handle **new types of GUI faults** that current GUI standard testing tools cannot detect. While GUI testing approaches have demonstrated their ability in finding several kinds of failures (*e.g.*, regressions, crashes) on standard GUIs [NRBM14, CHM12, APB+12, MPRS12, NSS10], various kinds of faults, which are specific to the characteristics of recent developments of GUIs, have still need to be identified and studied.

[2] A GUI toolkit is a library (or a collection of libraries) that contains a set of widgets to support the development of GUIs for instance Java Swing, or JavaFX toolkits.

[3] WIMP stands for *Windows, Icons, Menus, and Pointing device.*

These faults are mostly stemming from post-WIMP GUIs, which have more embedded HCI (Human-computer interaction) features such as *direct manipulation*, and *feedback* [HHN85, Nor02, BL00, BB10]. In WIMP GUIs, checking whether a *standard widget* is correctly activated can be done easily using regression testing techniques [Mem07]. In post-WIMP GUIs, however, errors show up in *ad hoc* widgets and their interactions, and the graphical nature of these GUIs (*e.g.*, data rendering). Thus, testing such GUIs requires new testing techniques.

Second, like any code artifact, GUI code should be analyzed statically to detect implementation defects and design smells. Several bad design decisions can introduce design smells such as design principle violations (*e.g.*, OO principle: encapsulation), *ad hoc* software evolution, and non-use of well-known best practices (*e.g.*, patterns [GHJV95], guidelines [Sun01]). Indeed, the presence of design smells may indicate a fault-prone code [LS07, HZBS14]. For example, the design smell "*God method*"[4] has been associated with the class error-probability [LS07].

Few efforts, however, have been done to investigate specific problems that arise from the development of GUI systems. To develop GUIs, software engineers have reused architectural design patterns, such as MVC [KP88] or MVP [Pot96] (Model-View-Controller/Presenter), that consider GUIs as first-class concerns (*e.g.*, the View in these two patterns). These patterns clarify the implementations of GUIs by clearly separating concerns, and thus minimizing the *"spaghetti of call-backs"* [Mye91]. Yet, the use of these patterns does not ensure that a GUI code will not be affected by development problems, which become more apparent in large and complex GUIs [Kar08]. GUI development thus poses a challenge to the GUI quality: **identifying design smells that degrade GUI code quality**. We call *GUI design smells* that decisions affect negatively the code responsible for GUI behavior. These smells are particular recurrent issues that limit the maintenance and evolution of GUI systems. Furthermore, deciding whether a GUI code contains design smells is not a trivial task due to the absence of GUI metrics to smell them. Software engineers have adapted well-known metrics such as *cyclomatic complexity* or *code duplication* to measure the GUI complexity. Also, cyclomatic complexity has been correlated to the number of faults found in the source code [KMHSB10]. So, we need to study which kinds of development practices affect the quality of GUI systems and which kinds of maintenance operations are required to correct them.

In this thesis, we address these two challenges by first providing a conceptual framework for GUI testing, *i.e.*, a GUI fault model, that covers the specificities of recent GUI developments. A GUI fault model must provide clear definitions of the different kinds of faults a GUI testing technique should look for. It then serves both as an objective to develop testing techniques and as a baseline to evaluate testing techniques. We address the second challenge through the development and empirical assessment of a novel static analysis technique, which specifically targets GUI code and focuses on one bad design practice that we call *Blob listener*. The objectives to improve the testing and maintenance of GUIs are listed as follows.

[4] Also called *long method*. It is a method that has grown too large [Fow99].

*GUI testing*:

- Provide a conceptual framework for GUI testing, which integrates the characteristics of GUI development paradigms;

*GUI maintenance*:

- Investigate development practices that characterize new GUI design smells, and develop a static analysis to automatically detect the presence of the critical ones.

## 1.3 Contributions

This thesis establishes two core contributions in the research of testing and maintenance of GUIs. We detail these contributions below.

1. **GUI fault model and its empirical assessment**: We have proposed a *GUI fault model* based on HCI concepts to identify and classify GUI faults. These faults concern WIMP and post-WIMP GUIs and they are described at two levels: user interface and user interaction. Our fault model tackles dual objectives: 1) provide a conceptual GUI testing framework against which GUI testers can evaluate their tool or technique; and 2) build a set of benchmark mutations, *i.e.*, *GUI mutants*[5], to evaluate the ability of GUI testing tools to detect failures for both WIMP and post-WIMP GUIs.

   To evaluate our fault model, we have conducted three empirical studies. In the first study, we assess the relevance of our fault model *w.r.t.* faults currently observed in existing GUIs. The goal is to state whether our GUI fault model is relevant against failures found in real GUIs. We have successfully classified 279 GUI-related bug reports of five highly interactive open-source GUIs in our GUI fault model. In the second one, we have evaluated the coverage of two well-known GUI testing tools to detect real GUI failures previously classified into our GUI fault model. These tools differ in how they build the GUI model, from specification of the SUT or from own SUT by reverse engineering. We have observed some faults related to standard widgets that GUI testing tools cannot detect such as GUI failures regarding the auto-completion feature or the data model. In the last study, we have executed these GUI tools against 65 GUI mutants derived

[5] All these mutants and the original version are freely available in `https://github.com/arnobl/latexdraw-mutants`

from our fault model. These mutants are planted into a highly interactive open-source system, and they concern both standard widgets and *ad hoc* widgets. This evaluation have demonstrated that GUI testing frameworks failed at detecting 43 out of 65 mutants, and most of them are related to *ad hoc* widgets and their user interactions.

The above experiments allowed us to study the current limitations of standard GUI testing frameworks. Thus, we have demonstrated why GUI failures that come from recent developments of GUIs were not detected by these frameworks. As a secondary contribution, we have introduced a richer User Interface Description Languages (UIDL), which is based on the concept of interaction-action flow graph (IFG), to tackle these limitations. We have presented the initial feedback from the use of this approach in two case studies: an industrial project and on an open-source interactive system. The preliminary results highlighted the capability of the IFG produces test cases able to detect some defects in advanced GUIs.

2. **GUI code assurance quality**: We propose a novel static analysis technique to automatically detect one specific *GUI design smell*, called *Blob listener*. We identified and characterized *Blob listeners* as a new type of GUI design smell. It occurs when a GUI listener can produce more than one GUI command. A GUI command is a set of statements executed in reaction to a GUI event triggered by each widget. This tool currently focuses on detecting *Blob listeners* in Java Swing software systems. We focus on the Java Swing toolkit because of its popularity and the large quantity of Java Swing legacy code. Also, `InspectorGuidget` leverages the Eclipse development environment to raise warnings in the Eclipse Java editor when it suspects the presence of *Blob listeners*.

   To design this tool, we first have conducted an empirical study to identify the good and bad practices in GUI programming. We examine in detail the GUI controllers code of open-source interactive systems stored on 511 Web code hosting services. These repositories contain 16 617 listener methods and 319 795 non-listener methods. The results pointed out a higher use of conditional statements (*e.g.*, *if*) in listeners used to manage widgets. Second, we identify several *good and bad coding practices* in GUI code. The critical bad development practice is the *Blob listeners*. We have observed that the use of *Blob listeners* in real GUI implementations complexifies the GUI code by, for instance, degrading the GUI code quality or introducing GUI faults. Third, we propose an algorithm to statically analyse GUI controllers and detect *Blob listeners*. Last, the ability of `InspectorGuidget` in detecting the presence of *Blob listeners* is evaluated on six highly interactive Java systems. `InspectorGuidget` have successfully identified 67 out of 68 *Blob listeners*. The results have exhibited a precision of 69.07 % and a recall of 98.81 %. Our experiments also show that 5 % of the analyzed GUI listeners are *Blob listeners*.

The above contributions have been published [Lel13, LBB15, LBBC15] or submitted for publication [LBB+16]. The preliminary challenges of testing interactive systems

appeared in [Lel13]. The work on fault model is published in [LBB15]. The kinds of GUI faults have been identified by performing a complete research study based on the HCI concepts and real bug reports. We also demonstrate the weakness of standard GUI testing tools at detecting several GUI failures by building a set of benchmark mutations derived from our fault model. The current limitations of GUI testing frameworks is published in [LBBC15]. In this work, we explain why standard GUI testing tools fail in testing advanced GUIs. We also present how a user interface description language, dedicated to advanced GUIs, can be applied on GUI testing to overcome that limitations.

The work on code assurance has been submitted [LBB+16]. In this work, we present the empirical study to investigate coding practices that affect Java Swing GUIs. We also present and evaluate the static analysis approach to automatically detect the presence of *Blob listeners*.

## 1.4 Overview of this thesis

The remainder of this thesis is structured as follows.

**Chapter 2** gives an introduction of GUI design by two generation of GUIs: WIMP *vs.* post-WIMP. This chapter also details the main concepts of human-computer interactions that must be considered in GUI testing. We then present the GUI V&V techniques (*e.g.*, defect classification schemes, dynamic and static analysis) and how the current approaches leverage them to provide a fully GUI testing.

**Chapter 3** presents an original, complete fault model for GUIs. The GUI faults are explained in detail through two levels: user interface and user interaction. This chapter also gives several examples of GUI failures originated by that faults. The empirical studies performed to assess the fault model are described in detail. We also present the limitations of standard GUIs testing frameworks in testing more advanced GUIs. We end this chapter by presenting the concept of interaction-action-flow graph to tackle these limitations.

**Chapter 4** starts showing the coding practices that may affect the GUI code quality. These practices are analyzed through an empirical study of GUI controllers through 511 code repositories. We then introduce the critical bad coding practice that we identified and characterized as a new GUI design smell. Next, we present an automated static analysis approach to detect this smell in GUI controllers called *Blob listener*. The empirical assessment of our approach's ability at detecting *Blob listeners* in six large, popular Java applications is presented in detail. We also propose the good coding practices to avoid the *Blob listeners*. This chapter ends with ongoing work towards refactoring *Blob listeners*.

**Chapter 5** presents the main achievements of this thesis and the future research. As a future research, we present simple examples of bad coding practices specific for

Java Swing toolkit that we identified. Such practices may be used to develop a kind of findbugs or PMD extension that targets GUIs.

# Part I

# State of the Art and Related Work

# Chapter 2

# State of the Art and Related Work

This chapter introduces the main concepts of GUI design to illustrate the new challenges for GUI testing, and presents the limitations of current state of the art technique. First, we present the characteristics that compose a GUI and explain them by the two generations of GUIs (Section 2.1). Then, we detail the interactive features that complexify how to develop and test GUIs. Second, we present the existing archictectural design patterns used to develop GUIs (Section 2.2). Last, we present the seminal GUI V&V techniques applied to support GUI testing (Section 2.3). We also explain how the current solutions leverage these techniques to provide an automated GUI testing process and point out their drawbacks.

## 2.1 GUI Design

GUIs are the essential part of an interactive system. They are composed of graphical interactive objects, called widgets, such as buttons, menus, drawing areas of graphical editors. These widgets are laid out following a specific order in their hierarchy. Users interact with widgets by performing a *user interaction* on a GUI. A user interaction is composed of a sequence of events (mouse move, mouse release *etc.*) produced by input devices (mouse, keyboard *etc.*) handled by users. Such an interaction produces as output an *action* that modifies the state of the system[1]. For example, the user interaction that consists of pressing the button "*Delete*" of a drawing editor produces an action that deletes the selected shapes from the drawing.

The graphical elements displayed by a widget are either purely aesthetics (fonts, *etc.*) or presentations of data. The state of a widget can evolve in time having effects on its graphical representation. For example, the widget properties (*e.g.*, visibility, position) have discrete values that may be modified during the execution of a GUI. Thus, a GUI state is a set of information collected from the states of all widgets while a user interaction is performed.

[1]The action may affect only the internal state of the system and thus its resulting may not be immediately visible by the user. For example, pressing the button *"Save"* in a text editor.

In the next subsection, we present the two generations of graphical user interface styles: WIMP and post-WIMP GUIs.

### 2.1.1 Types of GUIs

GUIs become more adapted to incorporate new interactions styles. This trend is reified by the two generations of graphical user interfaces styles: **WIMP** and **post-WIMP** GUIs.

**WIMP** (Windows, Icons, Menus, Pointer) GUIs [vD00, RPV12] emerged in the 70's aiming at being more user-friendly and easy to use[2] compared to the command lines interfaces. However, they only gained popularity with Macintosh in the 80's, and remained dominant for more than two decades [vD97]. WIMP GUIs are composed of *standard* widgets (*e.g.*, buttons, menus, text fields). Such widgets enable interactions composed of a single input event, we call *mono-event interactions*, based on the mouse and keyboard devices. The standard widgets work identically on many GUI platforms. For instance, interacting with a button has the same behavior on various platforms: *pressing* the left mouse button on it while a mouse is positioned. Also, the *standard widgets* have a pre-established set of properties (*e.g.*, position, size), which have predefined values (number, text).

Figure 2.1 shows a basic WIMP GUI. In this example, the standard widgets are represented by the pull-down menus "*File*" and "*Edit*", where interactions are performed through their menu items; the buttons *undo* and *redo* represented by the two *icons* grouped in the *toolbar* along the top; and the main *text area*. The interactions can be performed using a mouse. An example of a WIMP interaction is pressing the button "*undo*" in the toolbar, which undoes the text typed in that text area.

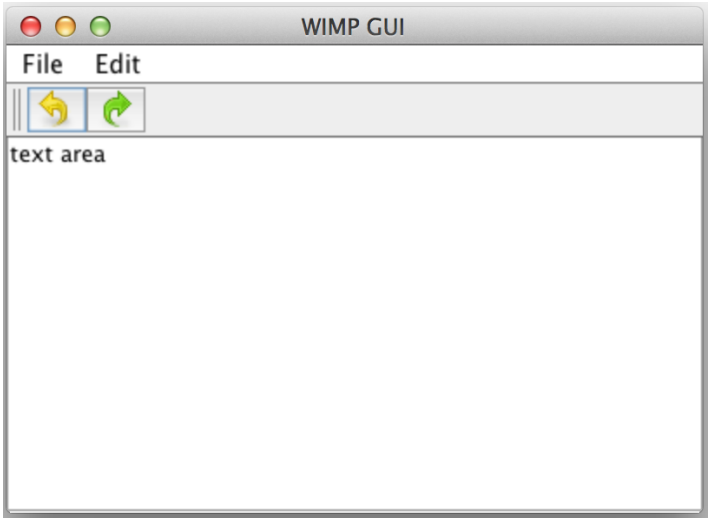


Figure 2.1: Example of a small WIMP GUI

[2] A user opens a window and then clicks on an icon by a pointer.

**Post-WIMP GUIs** go beyond the mere use of mice and keyboards to fit to new input devices such as tactile screens, gyroscopes, finger/eye trackers. They are presented in several domains such as ubiquitous and mobile computing, and virtual reality. These domains leverage the novel human-computer interaction techniques to develop highly interactive GUIs with more "*real world*" interactions [JGH+08]. van Dam proposed that a post-WIMP GUI is one "*containing at least one interaction technique not dependent on classical 2D widgets such as menus and icons*" [vD97]. Thus, post-WIMP GUIs rely more and more on *ad hoc* widgets and their complex[3] interactions.

*Ad hoc* widgets are mainly developed for a specific purpose such as GUIs in avionics systems [FFP+13]. These widgets are dynamic and their underlying data are not "predefined" like WIMP GUIs. Notably, objects in post-WIMP GUIs have more complex data structure such as geometric objects in 2D (*e.g.*, vector-based graphics systems) or 3D systems. These objects are more dynamic since they evolve as interactions are performed. For example, 3D objects change their state of appearance as the user interacts with them. Besides, *ad hoc* widgets allow users to interact more directly with objects (*e.g.*, data). We call such interactions *multi-event interactions*. They are composed of multiple events (in opposition to mono-event interactions) produced by one or more input devices. So, post-WIMP interactions are either sequential (*i.e.*, multiple events) and simultaneous (*i.e.*, multiple interactions). For example, a *multimodal* interaction[4] can be represented by a finite state machine, where each transition is an event produced by an input device [BB10].

Figure 2.2 illustrates a graphical post-WIMP editor called CNP2000 [BLL00, BL04]. This editor was redesigned for manipulating colored Petri nets through complex interactions techniques. The GUI of CNP2000 is based on toolglasses, marking menus, a *floating* palette[5] and bi-manual interaction to manipulate objects. A bi-manual interaction is a variant of direct manipulation that requires both hands to perform an action over a GUI (*e.g.*, zooming or resize a GUI object).

### 2.1.2 Seminal HCI Concepts

Identifying GUI faults requires an examination in detail of the major HCI concepts. In this subsection we detail these concepts to highlight and explain in Chapter 3 the resulting GUI faults.

*Direct manipulation* aims at providing users the feeling of visualizing and manipulating directly data objects of the system, as they could do in the real world [Shn83, HHN85]. Such a manipulation also recommends that user interactions that can be used to handle data objects can be inspired by the physical world, for permitting:

[3] These interactions are more complex from a software engineering point of view. From a human point of view they should be more natural, *i.e.*, more close to how people interact with objects in the real life.

[4] *"A multimodal HCI system is simply one that responds to inputs in more than one modality or communication channel (e.g., speech, gesture)"*[JS07].

[5] A *floating* palette differs from traditional palettes since it does not have the notion of selection, instead it floats above other windows.

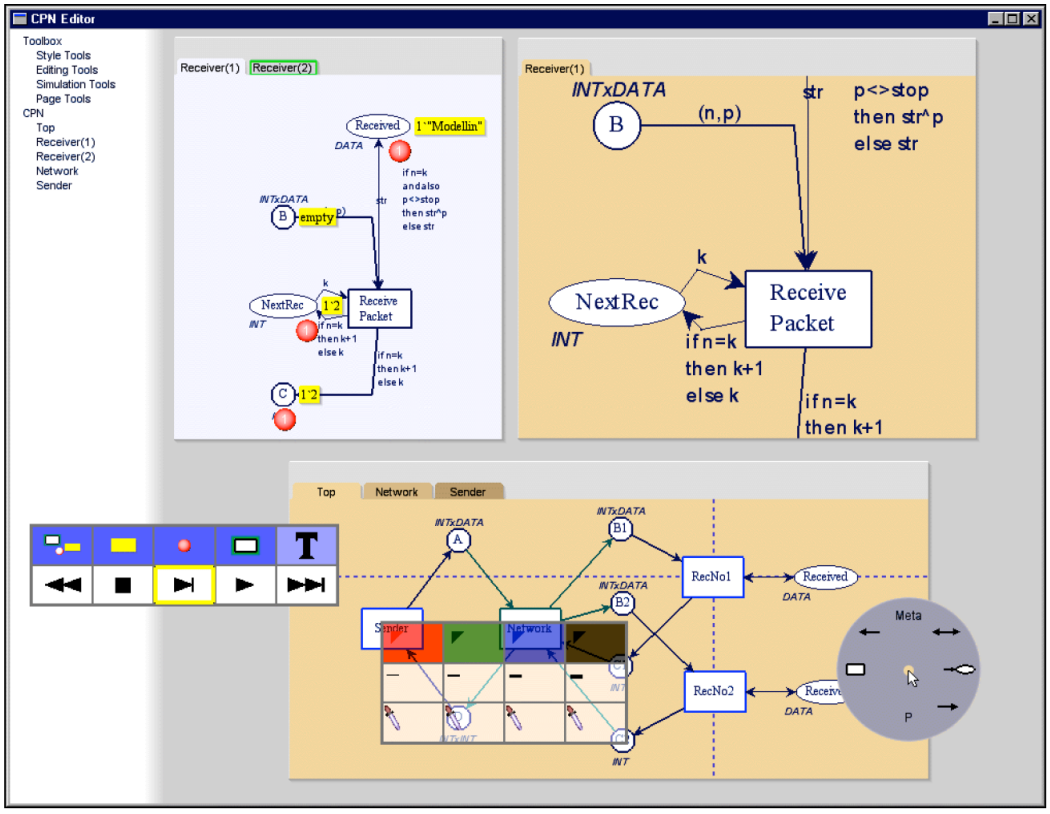


Figure 2.2: The post-WIMP editor CNP2000 [BL04]

- beginners to quickly learn the basic features of the system;
- experts to quickly perform multiple actions;
- all users to perceive in real-time the results of their ongoing interaction within the system.

Using direct manipulation also prones the following properties:

- data objects have to be continuously represented;
- physical interactions have to replace WIMP ones;
- actions, that modify the state of the system, have to be incremental and reversible, and have to provide users with feedback to let him monitor the progress of the ongoing action.

An example of direct manipulation is the drawing area of drawing editors (for instance the one depicted in Figure 2.2). Such a drawing area represents shapes as 2D/3D graphical objects as most of the people define the concept of shapes. Users can handle these shapes by interacting *directly* within the drawing area to move or scale shapes using advanced interactions such as bi-manual interactions. Direct manipulation is in

opposition to the use of standard widgets that bring indirection between users and their objects of interest. For instance, scaling a shape using a bi-manual interaction on its graphical representation is more direct than using a text field. So, developing direct manipulation GUIs usually implies the development of *ad hoc* widgets, such as the drawing area. These *ad hoc* widgets are usually more complex than standard ones since they rely on: advanced interactions (*e.g.*, bi-manual, speech+pointing interactions); a dedicated data representation (*e.g.*, shapes painted in the drawing area). Testing such heterogeneous and *ad hoc* widgets is thus a major challenge.

Another seminal HCI concept is *feedback* [HHN85, Nor02, BL00, BB10]. Feedback is provided to users while they interact with GUIs. It allows users to evaluate continuously the outcome of their interactions with the system. Direct manipulation relies on feedback to provide users with continuous representations of the data objects of the system. Feedback is computed and provided by the system through the user interface and can take many forms. A first WIMP example is when users move the cursor over a button. To notify that the cursor is correctly positioned to interact with the button this changes its shape. A more sophisticated example is the selection process of most of drawing editors that can be done using a Drag-And-Drop (DnD) interaction. While the DnD is performed on the drawing area, a temporary rectangle is painted to notify users about current selection area.

Another HCI concept is the notion of *reversible actions* [Shn83, HHN85, BB10]. The goal of reversible actions is to reduce user anxiety by about making mistakes [Shn83]. In WIMP GUIs, reverting actions is reified under the undo/redo features usually performed using buttons or shortcuts that revert the latest executed actions. In post-WIMP GUIs, recent outcomes promote the ability to cancel actions in progress [ACP12]. In such a case, users may cancel an interaction while it is executed, when it only changes the state of the view. For instance, moving a graphical object (*e.g.*, a shape) to a new position can be reverted before the object reaches that position. Thus, this feature also allows widgets to be more susceptible to direct manipulation.

All these HCI concepts introduced in this subsection are interactive features that require significant efforts to develop and test GUIs. Software engineers have paid special attention to integrate these features in GUIs. Such an integration has been done through complex pieces of code, which must be thoroughly analyzed and tested to detect faults. While GUI implementations rely on existing GUI architectural design patterns to minimize recurrent development problems and thus provide a higher quality code, GUI testing techniques have adapted existing testing techniques to provide GUI solutions that ensure the reliability of GUIs.

We thus present in the next sections the GUI architectural design patterns used to develop GUIs and the GUI V&V techniques applied to GUI testing.

## 2.2 GUI architectural design patterns

The software development of GUIs relies, as any software intensive-system [MEH01], on specific architectural design patterns that organize the GUI components and describe

how they interact with each other. This section gives a brief overview of the mainstream architectural design patterns used to develop GUIs.

The core decomposition that follows all the GUI architectural design patterns is the separation between the data model of the system (*aka.* the functional core) and their representations (the views or user interfaces). This explicit separation permits the model to be independent of any view. So, multiple views can be developed for a single data model. Views can be added, removed, or modified without having an impact on the model.

Based on this model-view separation, multiple GUI architectural design patterns have been proposed to complete this separation with details. The most well-known architectural design pattern is certainly the *Model-View-Controller* design pattern (MVC) [KP88]. MVC organizes an interactive system in three main components supplemented with communications between them, as depicted by Figure 2.3. The model corresponds to the data model of the system. The view is the representation of this data model. The controller is the component that receives events produced by users while interacting with the view, treats them to then modify the model or the view. On changes, the model can notify its observers (in our case the views) about the changes to be updated.


Figure 2.3: The MVC architectural design pattern

The transmission of events from a view to its controller usually relies on an event-driven programming pattern: an interaction produced through the view by a user using an input device (*e.g.*, pressing a button using a mouse) produces an GUI event (*e.g.*, a mouse event). This GUI event is then transmitted to the controller of the view that receive it through a listener method (*aka.* a handler method). For instance, Listing 2.1 illustrates a Java Swing listener method *actionPerformed* implemented in a mere controller *AController*. This listener receives events (*ActionEvent*) produced while interacting with buttons or menus. This event can be then treated by the controller in this method to perform some action.

```
1 class AController implements ActionListener {
2    @Override public void actionPerformed(ActionEvent e) {
3     //...
4    }
5 }
```

Listing 2.1: Example of a GUI controller in Java Swing

A second GUI architectural design patterns is the *Model-View-Presenter* pattern

(MVP) [Pot96]. MVP shares many similarities with MVC. The main differences are the renaming of the controller as a presenter and the communications between the three components, as depicted by Figure 2.4. The presenter is the core of the MVP pattern since the model does not notify its views anymore. The presenter manages the process of the events produced by the views and updates these last. This change permits the model to be fully independent of observers (views). MVP is mainly used to develop Web applications where the model remains on the server, the view on the client, and the presenter cut in two parts: one part on the client and another one on the server). In this case the presenter manages the communications between the client and the server. Implementations of MVP mainly rely on the use of listeners/handlers to treat GUI events.


Figure 2.4: The MVC architectural design pattern

The differences between MVC and MVP are, however, not so clear in practice since these patterns provide few details about how to implement them. Many variations and interpretations on MVC exist so that one may already be using MVP under the guise of MVC. Besides, a second version of MVC, called MVC2, has been proposed and matches the MVP definition[6]. Beaudoux *et al.* discuss the distinctions between MVC and MVC2 [BCB13]. MVC2 evolves MVC to allow the definitions of complex relationships between models and views such as data binding mechanisms. Both patterns, however, do not provide mechanisms for separating the interactions (*e.g.*, DnD) on presentations from actions (*e.g.*, Create) on the domain objects. In such a case, a mechanism should be provided for transforming a user interaction into an action that has effect on the presentation and/or on the domain objects.

Other industrial and academic GUI architectural design patterns have been proposed. We focus on two of them related to MVC and MVP that rely on event-driven programming. The *Model-View-ViewModel* pattern (MVVM) is the core component of the WPF (*Windows Presentation Foundation*) toolkit developed by Microsoft [Smi09]. MVVM can be summarized as the MVP pattern supplemented with WFP features, such as data binding. Regarding academic patterns, Malai [Blo09, BB10] can be viewed as an MVC or MVP pattern supplemented with two other components: interaction and action. These components respectively permit developers to explicitly define user interactions as finite-state machines and actions as reversible GUI commands.

The use of GUI architectural design patterns clarifies the implementations of GUIs by clearly separating concerns, and thus helps developers to avoid some problems such as the "spaghetti of call-backs" [Mye91]. The GUI implementations, however, still

[6] `http://www.javaworld.com/article/2076557/java-web-development/understanding-javaserver-pages-model-2-architecture.html`

are affected by development issues which may be accentuated by the multiplication of input devices (tactile screen, gyroscope, *etc.*), the diversification of the platforms (tablet, mobile, *etc.*), and the interactive features (recall Section 2.1.2). We investigate the development practices that affect the controllers of a GUI in Chapter 4.

## 2.3 GUI V&V techniques

GUI verification and validation (V&V) is the process of evaluating GUI software artifacts to verify whether they satisfy the requirements, or ensure that they are bug-free. One may note that GUIs have several specificities that complexify how to test them (recall Section 2.1). For example, testing WIMP GUIs focuses basically on testing *standard widgets*. Testing post-WIMP GUIs, however, focuses on both *ad hoc* widgets and their multi-event interactions. Besides, testing a GUI is able to find problems that arise through GUIs but may not be GUI failures. As Memon wrote GUI testing is "*not only how much of the code is tested, but whether the tested code corresponds to potentially problematic user interactions*" [Mem03].

There are several GUI V&V techniques (*e.g.*, model-based testing, static analysis) to ensure the quality and reliability of GUIs. They have been developed to find failures that affect GUIs or to measure the GUI code quality. Table 2.5 illustrates some V&V techniques and their solutions for graphical user interfaces.

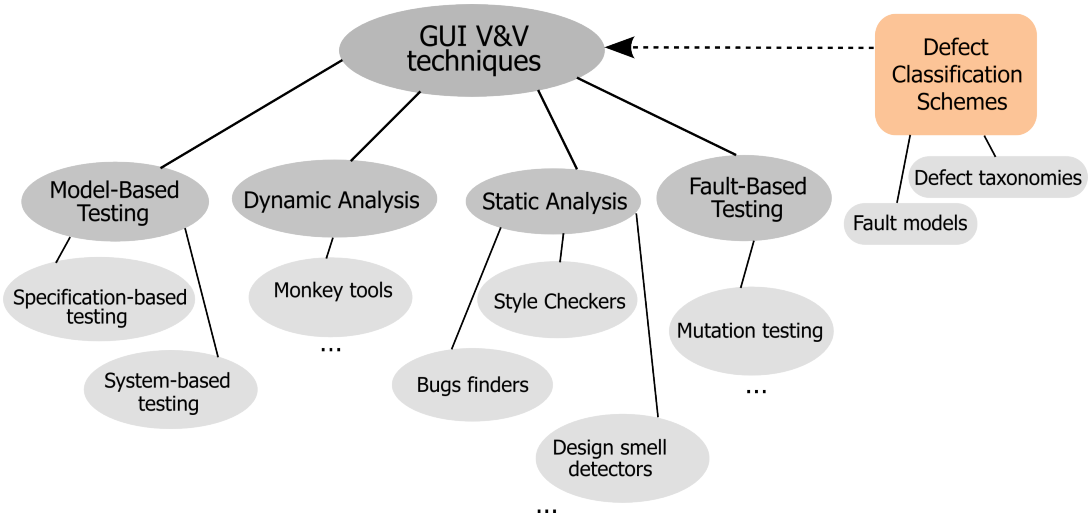


Figure 2.5: Examples of V&V techniques for graphical user interfaces

The most used technique to measure the quality and reliability of GUIs is *testing*. Testing GUIs aims at validating whether a GUI has the expected behavior: *Does the GUI conform to its requirements specification?* Indeed, GUI testing exercises each possible state of a GUI through its components looking for failures. Most of GUI testing approaches focus on automated GUI testing to provide an effective bugs detection

[Pim06, MPRS11, NRBM14]. For example, model-based GUI testing solutions rely on models to drive the test case generation. In such a case, the reliability of the GUI depends on the reliability of the models used for modeling and testing GUIs [BBG11].

The reliability of GUIs, however, is still affected by the low quality code. Static analyses have been thus proposed to measure the quality of a GUI code. They aim at detecting problems in the source code that may have effect on faults. Furthermore, V&V techniques are complementary, and similar GUI V&V techniques can be combined to achieve a fully GUI testing. For example, a model-based testing may use a dynamic analysis to build a GUI test model.

GUI V&V techniques can be supported by defect classification schemes. An example is *fault models* (Section 2.3.1.3) that are largely used as a start point in fault-based testing such as mutation testing. Mutation testing leverages fault models to implement a set of mutations that should be detected by their test suites. Thus, a complete test suite must detect all mutations derived from the fault model.

In the next subsections, we present seminal V&V techniques. This section is organized as follows. We first introduce the defect classifications schemes adapted for GUIs (Section 2.3.1). Then, we present the current GUI approaches to automate GUI testing such as model-based testing (Section 2.3.2) and dynamic analysis (Section 2.3.3). Last, we present the GUI solutions that leverage static analysis (Section 2.3.4).

### 2.3.1 Defect classification schemes

Testing techniques look for the presence of bugs[7] in a software system. One way to determine which defects should be overcome by such techniques is to introduce defect classifications. A defect classification scheme defines a set of defect categories, where each one covers a specific defect. Software defect classification schemes concern problems that affect software systems. They can be generic, describing problems related to the entire software development process, or software specific focusing on the product (*e.g.*, a mobile system) or a specific part of software (*e.g.*, GUIs). Most of the existing schemes have been proposed through two classifications: *defects taxonomies* or *fault models*. Both classifications have been used to support GUI V&V techniques.

In this subsection, we introduce the defect classifications schemes used for GUIs. We also present the limitations of theses schemes in covering GUI faults. The purpose is to clarify which kinds of faults the GUI V&V solutions should look for.

#### 2.3.1.1 The basic concepts

In this subsection, we present the definitions of fault, error, and failure that will be used throughout the next sections.

**Definition 2.1 (Fault)** *A fault is a static defect in a system [Wei10].*

**Definition 2.2 (Error)** *An error is a wrong internal state of a running system [Wei10].*

[7] We use the term *bug* to characterize any problem in a software such as fault/defect, error, or failure.

**Definition 2.3 (Failure)** *A failure is an incorrect external behavior of a system.*

A *static defect* is any incorrect behavior in a system component such as an incorrect instruction into a source code. IEEE *Std.* defines a *defect* as an imperfection in a product when this does not meet its specifications [IEE10]. We use the terms *defect* and *fault* as synonyms. A fault may exist into a system with no effect on its behavior, in this case a fault is said *dormant*, otherwise it is *active*. When a fault is *activated*, *i.e.*, that instruction is executed, it leads to a *wrong internal state* of a system (*e.g.*, erroneous program counter). If this wrong state is propagated outside a SUT, *e.g.*, it is observable by a user or an oracle, it is characterized as a *failure*. Thus, a fault can lead to one or more failures. Yet, a failure may be caused by user mistakes such as wrong input data.

When the aforementioned problems concern a GUI, the terms GUI fault, GUI error, and GUI failure are analogously defined. We present these definitions in Chapter 3.

#### 2.3.1.2 Defect taxonomies

Defect taxonomies are generic classification schemes since they focus on high-level problems that affect the software development process. Several defect taxonomies have been proposed in the literature [Bei90, CBC+92, Gra92, IEE10]. A comparison study of defect classifications taxonomies [ML09, VGH09] reveals that they share several commonalities such as some defect categories. A defect taxonomy is composed of categories, where each category is represented by a set of values that characterizes a defect. The relationships between such categories define the structure of a defect taxonomy (*e.g.*, orthogonal [CBC+92] when defect categories are independent).

The three defect taxonomies that have been adapted to cover GUIs are *Beizer's taxonomy* [Bei90], *Orthogonal Defect Classification* (ODC) [CBC+92], and *IEEE standard 1044* [IEE10]. We first introduce the general structure of these taxonomies and then we present the research studies based on them to cover GUIs.

**Bezier's taxonomy** is one of the first taxonomies to classify defects that was proposed by Boris Beizer. The defects are presented into eight categories as follows:

1. "*Functional Bugs: Requirements and Features*" concern defects caused by ambiguous, incomplete, duplicate, or overly specified requirements.
2. "*Functionality as Implemented*" categorizes defects through the incorrect implementation of requirements.
3. "*Structural Bugs*" focus on problems into the code structure such as unreachable code or wrong arithmetic expressions.
4. "*Data*" defects are related to the incorrect structure or manipulation of data.
5. "*Implementation and coding*" concern defects into the implementation (*e.g.*, typographical defects) and also defects that impact on the software maintenance such as standards violation in control-flow structure (*e.g.*, excessive *if-then-else* nesting).

6. "*Integration*" categorizes problems regarding the components integration such as an incorrect communication between internal and external interfaces.

7. "*System and Software Architecture*" focus on defects that affect the entire software system such as architectural errors.

8. "*Test Definition or Execution Bugs*" gather defects found in the definition, design, and execution of tests, and the data used to validate the system (*e.g.*, wrong test initialization, incomplete test cases).

Moreover, each above category is further break down into subcategories. This hierarchy structure specializes the defect categories to classify more precisely a defect. So, a defect is represented by a four digit number, where the first two digits concern the category and a subcategory, respectively. For example, the number 61xx classifies a defect within the category "Integration" and the subcategory "Internal interfaces" (see Table 5.1 in Appendix).

**ODC taxonomy** defines eight defect types: *function*, *interface*, *checking*, *assignment*, *timing/serialization*, *build/package/merge*, *documentation*, and *algorithm*. Also, the defect types can be associated with the software development phases (*e.g.*, design, code). The *interface* type concerns defects into the communication between software components (*e.g.*, modules, subsystems) or hardware (*e.g.*, device drivers). An example of *Interface* defect is *a wrong type of variable used in the parameter of a function call*.

**IEEE standard 1044** aims at classifying software anomalies. The term *anomalies* is used to describe any problem detected within the project, product, or a software life cycle. Also, IEEE *Std.* distinguish the terms *problem*, *defect*, *fault*, *error*, and *failure*. The terms *fault* and *defect* are not used interchangeably. For example, a defect is not considered as a fault if it is found by inspection or static analysis activities. Thus, IEEE *Std.* proposes a scheme that allows software engineers to categorize defect, faults, and failures, and their relationships. The defect categories are represented by a set of attributes such as *interface, logic, and data*. Similarly to *ODC taxonomy*, the root cause of the defect can be correlated with the phase of software cycle life (*e.g.*, requirements, designing, code, configuration). Also, defects into the *interface* type concern the same problem: communication between components. Table 2.1 illustrates the defects into this category.

Table 2.1: Examples of *Interface* defects presented by IEEE *Std.* [IEE10]

| **Attribute** | **Value** | **Example of Defects** |
|---|---|---|
| Type | Interface | Incorrect module interface design or implementation<br>Incorrect report layout (design or implementation)<br>Incorrect or insufficient parameters passed<br>Cryptic or unfamiliar label or message in user interface<br>Incomplete or incorrect message sent or displayed<br>Missing required field on data entry screen |

Although the aforementioned taxonomies aim at providing an extensive defect classification scheme, they present several shortcomings. One of this is the lack of low-level defect classifications, for instance, defect categories for covering specific concerns such as GUIs. Thus, several approaches have adapted existing defect taxonomies (*e.g.*, *Beizer's taxonomy* or ODC) or proposed new ones to cover GUIs. Besides, several GUI classifications focus on classifying GUI failures instead of GUI faults. We discuss these approaches below.

### GUI Defect classifications

The ODC extension to cover GUI defects was proposed by IBM Research [Res13a]. This extension defines 6 ODC triggers[8] that concern the visual aspects of GUIs. Table 2.2 shows the triggers (highlighted in blue color and italic) and the corresponding defect types. These types remain the same as described in ODC version 5.2 for *Software Design and Code* classification [Res13b].

Table 2.2: ODC triggers for graphical user interfaces [Res13a]

| Defect Removal Activities | Triggers | Defect Type |
|---|---|---|
| GUI Review | *Design Conformance* | Assignment /Initialization |
| | *Widget/Icon Appearance* | Checking |
| | *Screen Text/Characters* | Algorithm/Method |
| | *Input Devices* | Timing/Serialization |
| | *Navigation* | Interface/O-O Messages |
| | *Widget/GUI Behavior* | Relationship |

Likewise, Li *et al.* propose a new classification scheme called *Orthogonal Defect Classification for Black-box Defect* (ODC-BD) [LLS10]. This classification defines five high-level defect categories: *Function, System, Data, Interface, Specific*. The category *Interface* is divided into subcategories to cover several GUI defects. These subcategories are named according to specific GUI elements (*e.g.*, *window*, *title bar*, *menu*, or *tool bar*), or input devices (*i.e.*, mouse and keyboard).

An adaptation of Beizer's taxonomy (recall the beginning of Section 2.3.1.2) to include GUI-related defect categories is proposed by Brooks *et al.* [BRM09]. The extension creates new five subcategories that are related to GUIs: *GUI Defects*, *Software Documentation*, *User Documentation*, *Configuration Interfaces*, and *System Setup*. Table 5.1 (see Appendix) shows these subcategories (highlighted in blue color and italic) into Beizer's taxonomy. The subcategory "*GUI defects*" is included in the category "*Implementation Defect*" to classify defects concern both the graphical components of GUIs and the user interactions. Thus, the type of GUI defects, however, are not described and no GUI defect classification is proposed. Other approaches combine two or more defect taxonomies to identify faults. For example, Børretzen *et al.* [BC06] present a fault classification by combining IEEE *Std.* 1044 and ODC taxonomy. Similarly to

[8] "*A defect trigger is a condition that allows a defect to surface*" [CBC+92].

Brooks' work [BRM09], one category that focuses on GUI faults is added (*GUI faults*) but no GUI fault classification is proposed.

Other defect classifications focus on supporting the GUI testing automation. For instance, Xuebing has defined a small defect classification to drive a particular test case generation [Yan11]. This classification covers few types of GUI defects which are grouped in the three following categories:

1. *Functional defects* are defects found when events from functional objects (*e.g.*, buttons, menus) are invoked;

2. *Interactive defects* are defects that concern the interaction between a user and the SUT to exchange some information (*e.g.*, data editing);

3. *GUI adjustment defects* are defects regard the GUI adjustments objects, which are used to resize the GUI such as resizing, or scrolling adjustements.

The above categories are based on the most common standard widgets. Table 2.3 illustrates these categories according to their related widgets. A set of widgets is linked to a defect category (called "defect class" by the author). For instance, the category "*Functional defects*" is linked to widgets that have their results reflected in GUI properties (*e.g.*, buttons), whereas the category "*Interactive defects*" is related to widgets that receive inputs from the users (*e.g.*, combo boxes).

Table 2.3: Defects and their related objects presented by Xuebing [Yan11]

| **Defect Class** | **Related Object Type** |
|---|---|
| Functional defects | Button, Tool Button, Menu Item, List Item, Tree View, *etc.* |
| Interactive defects | EditBox, ComboBox, CheckBox, Spinner, TrackBar, *etc.* |
| GUI adjustment | Scroll Bar, Title Bar, Split Container, Tab Layouts, *etc.* |

### GUI failure classifications

In contrast to the defect-based classifications, several research studies focus on *failure-based taxonomy* [BM11]. This taxonomy is based on classifying failures instead of faults. Most of these studies focus on specific domains such as *mobile* [KMHSB10] or *safety-critical* [Lm04] or *specific GUIs* (*e.g.*, automotive GUIs [MKZD12]). For example, Maji *et al.* characterize failures for mobile operating systems [KMHSB10]. The failure categories are defined as *segments*. A segment is the localization (*e.g.*, a device component, a software system) where a failure occurred. For instance, a failure manifested in a *camera* is categorized in the *Camera segment*. Similarly, failures for other segments such as *Web*, *Multimedia*, or *UI softwares* (user interface) are categorized. Also, Zaeem *et al.* [ZPK14] have conducted a bug study for Android applications to automate oracles. They designed 20 bug categories and classified several failures into them. These categories have included some GUI issues such as *Rotation* (device's rotation), *Gestures*

(zooming and out), and *Widget*. Although, both research studies have investigated failures in a context that brings many advances in terms of interactive features, no GUI defect classification or discussion about these kinds of failures is presented.

Mauser *et al.* propose a GUI failure classification for automotive systems [MKZD12]. This classification is based on the three categories: design, content, and behavior. In the *Design* category, the failures refer to GUI layouts (*e.g.*, color, font, position). In the *Content* category, the failures are associated to data displayed such as text, animation, and symbols/icons. The failures in the *Behavior* category are caused by a wrong behavior of windows (*e.g.*, wrong pop-up) or widgets (*e.g.*, wrong focus). The authors focus on characterizing GUI failures based only on a small set of specific widgets designed for these kinds of GUIs. Furthermore, they do not consider failures that come from user interactions.

**Limitations of GUI defect classifications**

We identified several limitations of GUI defect classifications. The main shortcoming is that they are not properly defined to describe real GUI defects. The reasons are twofold. First, some classifications propose GUI defect categories that do not describe their types of GUI defects. For example, Børretzen *et al.* [BC06] and Brooks *et al.* [BRM09] have extended defect taxonomies to cover GUIs by including a single category called "*GUI defects/faults*", where the kinds of GUI defects are not described. Second, other classifications describe the GUI defects based on a set of standard widgets or input devices [LLS10, Yan11]. For example, ODC-BD [LLS10] proposes a defect category called *Mouse*. So, *does a fault classified into this category refer to an interaction, an action or an input device?* Also, these kinds of categories are more the case of GUI failure-based classifications, which identify the failures categories according to their localization. For example, the GUI failure that occurs when a user interacts with a widget is classified in the category *Widget* as proposed by Zaeem *et al.* [ZPK14].

The limitations of the GUI defect classifications are summarized below.

1. The defect categories are named according to a small and specific GUI components such as widgets (*e.g.*, button), or common input devices (*e.g.*, mouse);
2. The defect categories focus on standard GUIs, and thus faults that concern *ad hoc* widgets and their interactions are not covered;
3. The defect categories do not clearly separate defects that show up from different elements of a GUI system such as widgets, interactions, actions;
4. The defect categories do no cover interactive features (recall Section 2.1.2); and
5. The relationship between GUI defects and their GUI failures are not presented.

#### 2.3.1.3 Fault models

Fault models aims at being more accurate than a defect taxonomy. They describe how faults *come to be* and *how* and *why* they occur as failure [BM11]. Bochmann *et al.* [vBDD+91] define a fault model as:

**Definition 2.4 (Fault Model)** *A fault model describes a set of faults responsible for a failure possibly at a high level of abstraction and is the basis for mutation testing.*

This definition highlights two key points of fault models: 1. they provide different levels of abstraction; and 2. they qualify the effectiveness of testing techniques in terms of fault-detection. A higher level fault model describes faults covering both *physical faults* and *software faults* [vBDD+91]. The physical faults concern faults that affect the hardware components. One example is the well-known fault model *stuck-at model* that concerns physical faults within circuits in Digital Systems [MC71]. The software fault models provide a collection of faults that occurs in software systems. These faults can be structured according to the software assets (*e.g.*, specification, design, implementation). This allows software engineers to provide more detailed fault models and thus captures *"what is going wrong"* in a specific part of a system. Because of this specificity, faults models are widely applied to measure the fault-detection effectiveness of automated testing techniques. Binder wrote "*a fault model answers a simple question about testing techniques: why do the features called out by the technique warrant our effort?*" [Bin99].

One way to measure the fault coverage of test suites is to use a fault-based testing technique such as *mutation testing* (*aka.* mutation analysis). Mutation testing creates several mutants of a software system [JH11]. Each mutant contains a fault derived from a fault model, which is called a system's faulty version. A fault is seeded in the system's code as a mutant operator[9], which produces specific incorrect statements. The most common examples of mutants operators are: deletion/insertion/duplication of an statement, replacement of relational (*e.g.*, $>$, $>=$) or conditional (*e.g.*, &&, ||) operators. Once the mutants are planted, they are executed over a test suite. A mutant is killed when a test suite detects its corresponding failure.

### Type of fault models

Faults models have been largely used for supporting software testing techniques. They have been applied to describe software faults that concern the behavior of a system (*e.g.*, state transition models) or application domains such as object-oriented (OO) or concurrent systems. We present some examples of fault models below to:

- show that although GUIs are presented in several domains (*e.g.*, OO systems or critical interactive systems), the current fault models do not describe GUI faults; and
- illustrate how fault models should be structured to support for instance the fault-detection testing techniques.

**State transition fault models.** The GUI of a system is specified using User Interface Description Languages (UIDL). Such languages may leverage state transistion models to describe the GUI behavior. A state transition model is composed of states

[9] Also called mutation operators, mutagenic operators, mutagens, mutation transformations, and mutation rules [OU01, JH11].

and transitions between these states. Bochmann *et al.* presented several faults that affect FSMs and Petri nets [vBDD+91]. For example, an FSM is composed of states (initial, end, intermediate), events (input, output), and transitions. FSM faults refer to problems found in such a structure as detailed as follows.

- *output faults* when an output is not correctly provided;
- *transfer faults* when a transition goes to an incorrect state;
- *transfer faults with additional states* when a *transfer fault* leads to a non-specified state (*i.e.*, additional); and *additional or missing transitions* from one state to another.

Likewise, Petri nets faults refer to problems into Petri nets components (*e.g.*, arc, places, transitions). Some examples of such faults are: missing/additional input and output arcs, incorrect transition between the places.

FSMs and Petri nets have been used to describe GUI models [NPLB09, BB10]. GUI models can serve as a basis to build GUI test models. One example of an FSM model used to build GUI test models is presented in Chapter 3. If a GUI model describes correctly a GUI, the test model will produce the test cases correctly. So, the accuracy of a test model also depends on the accuracy of building correctly its representation.

**Object-oriented fault models** describe faults specific to object-oriented features. An example of OO faults is when OO principles are violated. Offutt *et al.* have proposed a fault model regards the misuse of inheritance and polymorphic principles [OAW+01]. Such faults are described independently of a programming language, but the manifestation of their failures is more prone to a specific language. Some examples of OO faults are listed below:

- *Inconsistent Type Use (context swapping)*;
- *State Definition Anomaly (possible post-condition violation)*;
- *Indirect Inconsistent State Definition*;
- *Incomplete Construction*;
- *State Visibility Anomaly*; *etc.*

Several research studies have used OO faults models to provide more specific mutant operators [Bin96, KCM00, MKO02]. The set of these operators is also referenced in the literature as **mutation fault models**. These models provide a collection of mutants operators to plant a specific fault. Ma *et al.* have presented six categories of mutation operators based on OO programming faults [MKO02]. One example is the incorrect use of *Inheritance* introduced by the operator *hiding variable deletion*. This operator removes a declaration of an overriding method, for instance, *commenting the line, where a variable is declared*. Thus, the related fault consists of the "incorrect variable access" since any reference to that variable will be accessed from the variable defined in the

parent (or ancestor) instead of its child. Also, the authors have proposed Java mutant operators. For example, the Java mutant operator "*this keyword deletion*" introduces the fault "*this keyword misuse*".

Likewise, Strecker *et al.* [SM08] have presented 12 mutant operators (called as *fault classes* by the authors) in Java code that affect GUI test suites. These mutants, however, do not characterize GUI faults but Java mutant operators (*e.g.*, class or method faults) that may affect a GUI. Table 2.4 illustrates some of them.

Table 2.4: Examples of fault types presented by Strecker *et al.* [SM08]

| Fault ID | Fault Classes | Examples |
|---|---|---|
| 1 | Modify relational operator | >, < , >=, <=, ==, != |
| 2 | Invert the condition statement | N/A |
| 3 | Modify arithmetic operator | +, -, *, /, =, ++, -, +=, -=,*=, /= |
| 4 | Modify logical operator | &&, \|\| |
| 5 | Set/return different *boolean* value | *true*, *false* |
| 6 | Invoke different (syntactically similar) method | N/A |
| 7 | Set/return different attributes | N/A |
| 8 | Modify bit operator | &, \|, &=, != |

**Aspect-oriented fault models** refer to faults that arise from aspect-oriented programming. AOP fault models describe faults that are distinct from OO and procedure-oriented programming [ABA04]. Thus, AOP faults are most likely to appear during the implementation of the cross-cutting concerns such as *pointcuts*, *join points*, and *advice* in AspectJ[10] language. Several AspectJ fault models have been proposed in the literature [ABA04, BA06, CTR05, DMM05]. Baekken *et al.* [BA06] describe several specific faults regarding the AspectJ *pointcuts*. One example is the *incorrect choice of primitive pointcut*, where one pointcut should be selected in place of another.

**Concurrency fault models** define several faults related to *concurrent systems*. The essential property of these systems is *concurrency*, which enables several computations executing simultaneously. Developers have faced several problems to implement such a property. One key issue is when the synchronization of concurrent systems is violated. Lu *et al.* [LPSZ08] have studied several *bugs* related to this problem. Each bug was analysed to identify the root cause and thus identify a bug pattern. The results of this analysis are presented into a fault model. This model is structured in three dimensions: *bug pattern*, *bug manifestation*, and *bug fix strategy*, which describe, respectively, the identified faults, their manifestation, and the strategy to fix them. Table 2.5 illustrates the kind of faults presented in the *bug pattern*.

**System-specific fault models** describe faults collected from empirical observations of an application domain. Such fault models differ from the aforementioned models since they do not focus on characterizing faults from programming features and languages. Instead, they are based on a system domain. One example is a preliminary

[10]It is AOP extension dedicated to Java programming language: `http://www.eclipse.org/aspectj/`

Table 2.5: Concurrency *Bugs* presented by Lu *et al.* [LPSZ08]

| Dimension | Category | Description |
|---|---|---|
| **Bug Pattern** | Atomicity Violation | The desired serializability among multiple memory accesses is violated (*i.e.*, a code region is intended to be atomic, but the atomicity is not enforced during execution). |
| | Order Violation | The desired order between two (groups of) memory accesses is flipped (*i.e.*, A should always be executed before B, but the order is not enforced during execution). |
| | Other | Concurrency bugs other than the atomicity violation and order violation. |

Web-specific fault model proposed by Ricca and Tonella [RT05]. This model is obtained by analysing the open-source bug repositories of Web applications written in different languages (*e.g.*, PHP, Javascript). The fault categories concern the *authentication problems*, *hyperlink problems*, *incorrect session management*, *incorrect generation of error page*, *etc.*

Other system domain that fault models have been used is critical interactive systems. In such a domain, the interface contains an amount of information which is shown in different displays by using a range of technologies (*e.g.*, LCD and CRT screens). This information must to be coherent in all displays to avoid failures provoked by the operator. One example is the interface of instrumentation and control (I&C) systems for nuclear power plants. Fayollas *et al.* have used fault models to increase the reliability of safety critical systems such as interactive cokpits in avionics systems [FFP+13]. The interactive cockpit user interfaces are composed events, display managers, and widgets (*e.g.*, push-buttons, radio buttons, edit boxes). These interfaces manage information from a human operator (*e.g.*, crew members) and from user applications (*e.g.*, aircraft systems). One way to validate such interfaces is to ensure that the information received from the operator (*e.g.*, input events) and the data received from the applications are processed correctly to be then visualized on GUIs. So, the three following failures must be avoided:

1. *Erroneous display* concern the transmission of wrong values to the display;
2. *Erroneous control* refer to the transmission of actions that differ from the events triggered by the operators;
3. *Inadvertent control* are related to the transmission of actions that are not performed by the operators.

To prevent these failures, the authors have considered a fault model that cover software faults (*e.g.*, design faults) and physical faults such as crash and transient faults.

By contrast, Pretschner *et al.* have proposed a **generic fault model** for quality assurance [PHEG13]. The authors qualified this model as *generic* since it aims at char-

acterizing fault models instead of their faults. They demonstrated how some existing fault models (*e.g.*, finite state machine models) can be instantiated from it.

What is the important about the aforementioned fault models is they are presented in several domains, which some of them (*e.g.*, critical interactive systems) are driven by GUIs. Although they describe specific faults that may result in failures (*e.g.*, crashes) observed by a GUI, none of them have presented or discussed GUI faults that may be the root cause for such failures.

In this subsection, we presented the defect classification schemes for GUIs and pointed out their drawbacks. We observed that the current GUI schemes bring several limitations to cover properly GUI faults. Such limitations help us to understand which kinds of faults existing GUI V&V techniques must be overcome. We thus present in the next three subsections the seminal GUI V&V techniques and how they are designed to support GUI testing. The purpose is to show the main drawbacks of GUI testing approaches.

### 2.3.2 Model-based GUI testing

Model-based testing (MBT) is a software testing technique that relies on models to describe the SUT behavior. MBT is widely applied to automate GUI testing. This subsection gives an overview of the basic MBT process applied for automated GUIs.

The GUI standard MBT process is illustrated in Figure 2.6. This process consists of obtaining *GUI test models* that are built from the specifications or directly from the SUT. Based on these models, *abstract test cases* are generated and then *concretized* to be executed against the SUT. The four MBT artifacts are detailed below.

- *GUI test models* are test models that describe the structure and behavior of a GUI of the SUT. They contain all possible sequence of user interactions. A GUI model, however, can be designed at different levels of abstractions: modelling standard GUIs based on single input events such as event-flow graphs (EFG) [Mem07]; modelling advanced GUIs based on multi-event interactions such as interaction-action flow graph (IFG) [LBBC15]; or modelling user scenarios to achieve a particular goal such as task models [SCP08].

- *Abstract GUI test cases* are non-executable test scripts generated by traversing a GUI test model *w.r.t.* a test adequacy criterion. A test adequacy criterion is defined to constitute an adequate test suite (*e.g.*, *what to test in a GUI?*). It helps to determine whether the test cases adequately check the SUT. An abstract GUI test case represents one possible sequence of user interaction described in the GUI test model.

- *Concrete GUI test cases* are executable test scripts concretized from the abstract test cases. A concrete test case contains information (*e.g.*, input data) that allows an abstract test case be executed over the SUT.

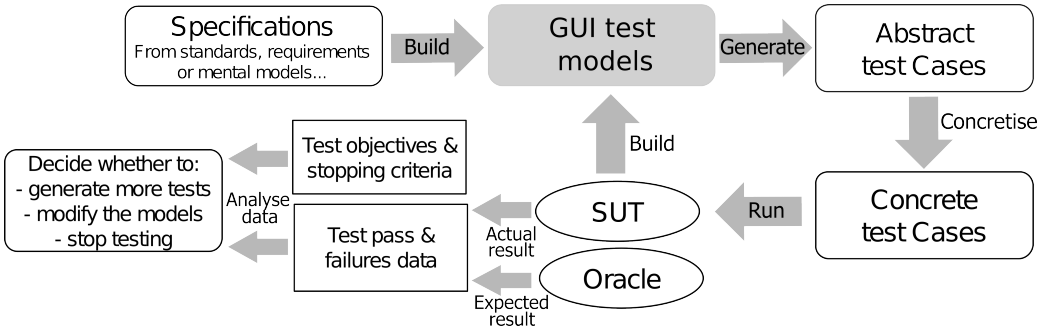


Figure 2.6: Model-based testing process. Adapted from [Lel13]

- *GUI Oracles* yield test verdicts by comparing the actual results of test scripts with the expected ones. A GUI oracle model can be obtained from the requirements specifications or from a stable version[11] of a GUI software.

GUI test models can be automatically extracted by reverse engineering from the SUT binaries [Mem07] (*i.e.*, using dynamic analysis). Such models of existing GUIs is effective for detecting crashes and regressions: the analysed GUI is then considered as a benchmark. In some cases, however, reverse engineering is not possible. For instance, the norms IEC 60964 and 61771, dedicated to the validation and design of nuclear power plants, require that: *developed systems must conform to the legal requirements*; *models must be created from requirements in that purpose* [IEC95, IEC10]. In this case, GUI models are designed manually from the requirements. The testing process then targets mismatches between a system and its specifications. Paiva proposed an automated MBT approach for GUIs based on the requirements specification [Pim06]. This specification is written in Spec#[12] language and then converted into a FSM to generate automatically GUI test cases.

Furthemore, GUI test models can be refined into GUI models, which are specified using a User Interface Description Language (UIDL). UIDLs are used to model GUIs in different domains (*e.g.*, critical interactive systems, rich interactive GUI systems). For example, Malai [BB10] and ICOs [NPLB09] architectures have been used to model rich GUIs, *i.e.*, post-WIMP GUIs, whereas GUITAR testing framework [NRBM14] uses its own UIDL that captures GUI structures (the widgets that compose a GUI and their layout). Examples of GUI test models are: state transition models (*e.g.*, FSMs or extended FSM such as Markov chain, or Petri nets), graphs (*e.g.*, EFG) or formal languages (*e.g.*, *Z* language). While Malai uses FSMs to represent GUIs, ICOs leverages Petri nets. Similarly, UML can also be applied to model user interactions using its state machine diagrams [OMG07].

In the current GUI testing approaches, test models are mainly event-flow graphs (EFG) [Mem07, APB+12]. EFGs contain all the possible sequences of user interactions

[11]This version (*aka. golden version*) is considered a fault-free version of a system, and it is used for doing regression testing [YCM11, SKIH11].

[12]This is a formal programming language developed by Microsoft Research [BLS05].

that can be performed within a GUI. One example is GUITAR that leverages EFGs to support an automated test cases generation. Figure 2.7 gives an general overview of GUITAR process. The first step consists of extracting the GUI structure (*e.g.*, widgets that compose a GUI and their properties such as visibility) by a *GUI Ripper* tool. Then, this structure is converted into a *GUI model*, *i.e.*, EFG. Next, the EFG is inferred to generate test cases by using *GUI tcgenerator* tool. So, *GUI Replayer* executes those test cases against the SUT. One example of a GUI test sequence to "*Draw a black square*" on the radio button demo shown in that Figure is: <*square*; *none*; *create shape*>.

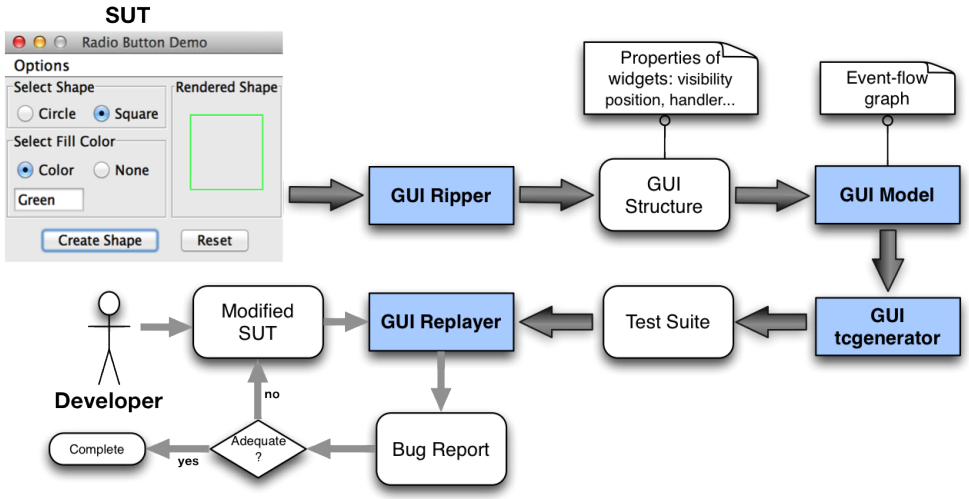


Figure 2.7: Overview of GUITAR process. Adapted from [Mem06]

The main shortcoming of the MBT approaches for GUIs is that they are based on models concerning standard GUIs. Such models will produce test cases that are limited to testing standard widgets and their *mono-event* interactions. An adaptation of these models is required to test advanced GUIs and thus to detect the new kind of GUI faults. The current limitations of the GUI testing frameworks dedicated to test standard GUIs (*e.g.*, GUITAR) and a promising GUI test model to test advanced GUIs will be discussed in Chapter 3. Furthermore, standard GUI test models (*e.g.*, EFGs) can be built from the SUT by using dynamic analysis. This brings other drawbacks that we discuss in the next subsection.

### 2.3.3 Dynamic Analysis

Dynamic analysis leverages the SUT execution to observe its external behavior. So, the only software artifact that can be used is the proper SUT (or its binaries). Due to the nature of GUIs, which require user interactions to produce actions, several approaches leverage dynamic analysis to exercise a GUI and thus test its external behavior to look for failures.

A GUI dynamic analysis technique explores the GUI of a SUT to extract all the

possible sequence of events[13]. The goal is to interact with a GUI through all the possible GUI states. To do this, each widget (*e.g.*, windows or buttons) is traversed and its events are triggered through the GUI. One example is the GUI Ripper step of GUITAR (see Figure 2.7) that dynamically explores a GUI. The information obtained from the GUI will be the basis to build a GUI test model. Three factors, however, should be considered to dynamically traverse a GUI [AFT+12]:

- *the way and the order the events are triggered*;
- *precondition of the SUT and its running environment*; and
- *a criterion to stop the GUI exploration.*

One problem that arises when the *order of events is not correctly triggered* is that: a GUI test model can produce *infeasible* test cases sequences. For example, a widget is disabled during the test execution when it should be enabled. In this case, the test may fail. Memon proposed an automated approach to repair the infeasible sequences [Mem08] generated by GUITAR.

Figure 2.8 illustrates the two artifacts generated by GUI Ripper tool: GUI tree on the top and its EFG model on the bottom. The *GUI Ripper* tool [MBN03] executes dynamically a SUT and extracts its GUI tree. This GUI tree represents the hierarchical GUI structure such as widgets (*e.g.*, $w_0$, ...,$w_8$, where $w_0$ represents the button "*Exit*") and their properties. These properties have discrete values (*e.g.*, integer or text) that compose a GUI state. The EFG represents the possible sequences of events behind the GUI structure. Each node of EFG is an event triggered by a widget. For example, the nodes named *create* and *exit* in the EFG represent the events triggered by the buttons "Create" and "Exit" in the GUI tree, respectively.

An extension of *GUI Ripper* to support Android platform, called *AndroidRipper*, is presented by Domenico *et al.* [AFT+12]. The process is similar to *GUI Ripping* but the GUI structure is represented by a state machine model. This model contains the set of GUI states and state transitions. The main goal is to find GUI crashes. Also, Takala *et al.* proposed a dynamic solution to build models of Android applications and generate online GUI test cases [TKH11]. The solution uses two separate state machines to build the model to facilitate its reuse on other device models. The first machine represents the state of the SUT and verifies it against the model. The second state machine uses the *keyword* method to capture the user interactions, for instance, *tapping* and *dragging* objects on the screen.

Mariani *et al.* proposed the *AutoBlackTest* technique to build a model generating test cases incrementally and automatically while interacting with a GUI of SUTs [MPRS11, MPRS12]. While interacting with the SUT, the GUI is analyzed to extract its current state. Then, the behavioral model is updated to select and execute the next action. Thus, the model is built incrementally and the test cases are identified. These test cases are refined and a test suite is then generated automatically.

[13]This exploration is also called "*crawling or ripping*" the GUI.

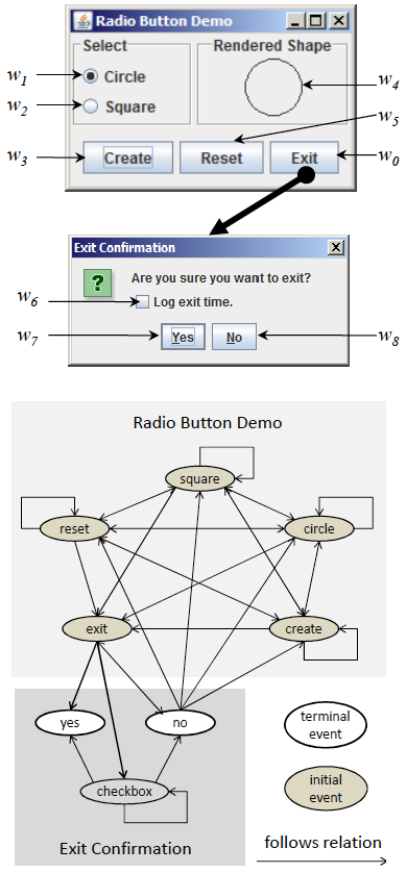


Figure 2.8: GUI Ripping artifacts presented by Memon *et al.* [MBNR13]

Although most of GUI dynamic approaches build test models to automate steps of a model-based GUI testing. Other solutions may generate sequences of events without creating a GUI test model. A typical example is the *Monkey tools* such as *UI/Application Exerciser Monkey* for Android systems. These tools interact with a GUI by randomly sending events to a device (or emulator) such as key pressures, touches pressures, or gestures. They are useful to find failures that result in crashes. Some model-based testing tools (*e.g.*, TEMA tools[14]) have reused *Monkey tools* to interact with a specific GUI (*e.g.*, Android GUIs) and thus extract the GUI structure.

The main drawback of GUI dynamic analysis techniques is that they require a correct SUT to extract GUI models. If a faulty SUT is used, its corresponding GUI model will be built incorrectly. So, the test cases generated from this model will consider GUI failures as a correct behavior. For this reason, GUI testing techniques that use dynamic analysis to support test cases generation have demonstrated their effectiveness during regression testing.

[14] `http://tema.cs.tut.fi/`

### 2.3.4 Static Analysis

While dynamic testing techniques exercise a SUT looking for the presence of failures, static analyses check whether a software artifact (which does not require the SUT execution) is fault or error-free. Thus, static analysis techniques are performed on several software artifacts such as the specifications (*e.g.*, model checking) or the source code (*aka.* static code analysis).

A *static code analysis* is a technique that examines the source code of a SUT without executing it. This technique can be applied for different purposes. One purpose is to infer the source code to detect bad coding practices such as *design smells*. GUI bad coding practices may be error-prone as standard bad coding practices. Another goal is to apply the static code analysis to locate faults directly in a source code, whether or not such faults may provoke a failure. Because of this, static analysis is *aka. variant of testing*.

In this subsection, we first give a brief introduction on the purpose of static code analysis tools. Then, we present the GUI static analysis approaches and how they target the quality of GUIs.

#### 2.3.4.1 Static Analysis tools

Static code analysis is usually supported by automated tools. There are several static code analysis solutions that vary in their scope from *design smell detection* [BMMM98, Fow99, MPN+12, SKBD14] to *maintenance and evolution of systems* [Sta07, APB+12, ZLE13]. Such solutions aim at solving several problems when the aspects of code quality (*e.g.*, understandability and changeability) are affected. The main solutions of static analysis tools are presented below.

**Style checkers** aim at assisting developers to write code that adheres to quality checkers. Quality checkers are a set of coding standards such as programming rules, naming conventions, and layout specifications. One example of a quality checker tool is *Checkstyle*[15]. This tool analyses the coding style and conventions of a Java code checking for *whitespace*, *method and line length*, *empty blocks*, *etc.*

**Code metrics** allow developers to calculate the structural attributes of a source code. Several studies has been done to establish metrics to measure the complexity of code since they are a strong indicator of software quality and maintenance cost. Code metrics are, for example, the number of lines of code (LOC), the McCabe's cyclomatic complexity (CC) [McC76], the cohesion metrics [RRP14].

**Code structures** are useful to understand different aspects of a source code such as control flow, data flow and data structures. Control flow structures concern the paths of an execution and can be thus used, for instance, to identify unreachable code. Data flow structures track a data item as it is accessed and modified by the code. One example of data-flow error is *assigning an incorrect or invalid value to a variable.*

[15] `http://checkstyle.sourceforge.net/`

**Bug finders** look for potential defects in the source code of a particular programming language. The goal is to raise warnings into a code where the system will behave incorrectly. Such tools are based on several rules (or patterns or idioms) to indicate often real defects. A well-known tool is *Find bugs*[16] that detects several errors in Java systems such as *infinite recursive loops*, *deadlocks*, and *catch exceptions*. Another tool to identify potential *bugs* and *bad programing practices* in Java code is PMD[17].

**Design smells detectors** look for bad design problems that degrade the code quality. A design smell (*aka.* code smells or bad smells) is any symptom in the source code that may indicate a design problem or a poor design choice. The characterization and detection of object-oriented design smells have been widely studied [BMMM98, Fow99, GPEM09, MGDLM10, KVGS11, MPN+12, PBDP+14, ZFS15]. The most relevant work about design smells is proposed by Fowler *et al.* [Fow99]. In this work, the authors describe 22 design smells and the refactoring strategies to correct them.

Note that some tools have more than one purpose, *i.e.*, they combine two or more solutions. For example, PMD looks for both *bugs* and *bad programing practices*. Similarly, *Style checkers* detect design violations that may be characterized as *design smells* [GSS15]. Several tools leverage that combination to cover a domain-specific. This is the case of static analysis tools for GUIs. These tools can leverage the code metrics (*e.g.*, CC) or code structures (*e.g.*, data flow structure) to formulate heuristics or to build/improve test models, that may also be used by detection strategies such as *design smell detections*.

#### 2.3.4.2 GUI Static Analysis tools

Table 2.6 gives an overview of several GUI approaches that use static analyses to achieve a specific purpose. They can be applied for supporting *GUI testing* by improving a GUI test model [SSG+10], or this obtained model can be used to detect *bad smells* [SCSS14]. We categorize those approaches according to the main purpose they serve.

Several approaches have extended GUITAR by using static analyses to extract information from the source code. The purpose is to improve **GUI testing** techniques by building a richer GUI test model, which will support the test case generation. For example, Arlt *et al.* use static analyses to identify and removes *redundant* test sequences in GUI test suites [APB+12]. They have extended GUI Ripper to infer dependencies between GUI event handlers[18] in the source code. These dependencies are included in a model called Event Dependency Graph (EDG). Similarly, Yang *et al.* implement a static analysis in a tool called *ORBIT* to extract a set of events from the source code of Android applications [YPX13]. Both static analysis approaches aim at building more complete GUI test models. Such models will produce more feasible test cases that will be able to find failures across different GUI domains.

[16] `http://findbugs.sourceforge.net/`

[17] `http://pmd.sourceforge.net/`

[18] These handlers are responsible for treating events fired by widgets, *aka.* (GUI) listeners [Sun01], or (GUI) controllers [LBB+16].

Table 2.6: A comparison of some GUI static analysis approaches

| Purpose | Approach | Tool | Subject Artifact | GUI Domain |
|---|---|---|---|---|
| GUI testing | Arlt *et al.* [APB+12] | Gazoo | GUI Model | Java and C# |
| | Yang *et al.* [YPX13] | ORBIT | Test Model | Android |
| | Silva *et al.* [SSG+10] | GUISurfer | GUI Model | Java Swing |
| GUI maintenance and evolution | Zhang *et al.* [ZLE13] | FlowFixer | GUI workflows | Java Swing |
| | Staiger [Sta07] | Bauhaus | GUI structures | C/C++ |
| | Zhang *et al.* [ZLE12] | WALA | Java bytecode | Java/Android |
| | Frolin *et al.* [OPM15] | AUREBESH | Inconsistencies | JavaScript |
| Design smells | Silva *et al.* [SCSS14] | GUISurfer | Dialog models | Any GUI |
| | Chen & Wang[CW12] | GTT | GUI test scripts | C/C++ |
| GUI metrics | Magel *et al.* [MA07] | - | Structural metrics | Any GUI |

Silva *et al.* use a static analysis to extract automatically the GUI behavior from the source code [SSG+10]. The static analysis is implemented in a tool called *GUISurfer*. The GUISurfer process is illustrated in Figure 2.9. First, a language-dependent parser is used to obtain the abstract syntax tree (AST) from the source code. A code slicing technique is used to interact with this AST and extract only GUI related data, which represent a GUI model. Then, a language independent tool receives this model and generates the GUI outputs files (*e.g.*, events, conditions, actions, and states). These files are transformed in behavior models such as FSMs to support GUI testing.

Other static analysis approaches focus on **GUI maintenance and evolution**. Grechanik *et al.* propose a tool called *REST* to identify test scripts that are affected when a GUI of a system evolve [GXF09]. Zhang *et al.* have extended *REST* to automatically repair broken workflows in Swing GUIs [ZLE13]. A workflow is broken when GUI widgets are replaced or shifted in a new version of the GUI. The authors combine random testing and static analysis to infer workflows from a user's point of view. The static analysis is limited to find methods related to a GUI action. Although a GUI listener can handle several actions[19], GUI actions presented in a same listener are not tackled. So, the static analysis approach was less effective. This work highlights the difficulty "*for a static analysis to distinguish UI actions [GUI commands] that share the same event handler [GUI listener]*". In this thesis, we tackle this problem by developing an approach to accurately detect GUI commands that compose GUI listeners.

Staiger also proposes a static analysis to extract GUI code, widgets, and their hierarchies in C/C++ software systems [Sta07]. This approach, however, is limited to find relationships between GUI elements and thus does not analyze GUI event handlers. Zhang *et al.* propose a static analysis to find violations in GUIs [ZLE12]. These violations occur when GUI operations are invoked by non-UI threads leading a GUI error. The static analysis is applied to infer a static call graph and check the violations. This technique is implemented on the top of *WALA* framework. An interesting finding reported by the authors is that "*GUI developers have already used design patterns, run-*

[19]We call GUI (or UI) actions as GUI commands, which will be detailed in Chapter 4.

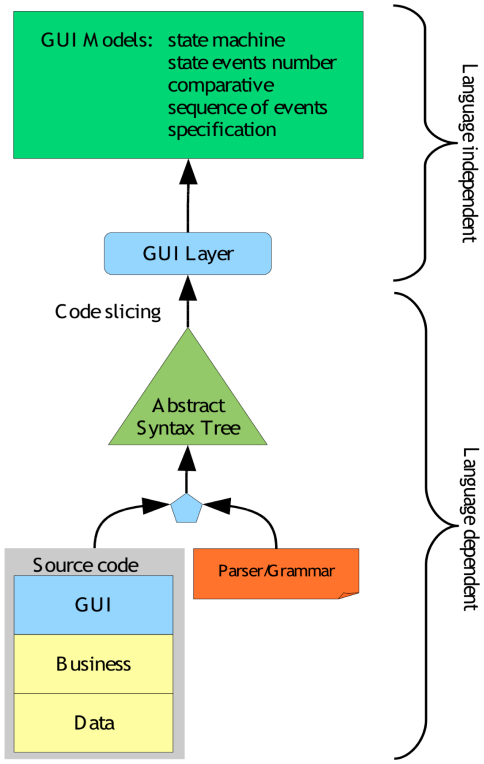


Figure 2.9: GUISurfer process presented by Silva *et al.* [SSG+10]

*time checks, and testing to avoid violating the single-GUI-thread rule. However, due to the huge space of possible UI interactions, hard-to-find invalid thread access errors still exist*". In our work, we tackle a GUI design smell that affects GUI controllers, which are based on GUI architectural patterns.

Frolin *et al.* propose an approach to automatically find inconsistencies in MVC JavaScript applications [OPM15]. GUI controllers are statically analyzed to identify consistency issues (*e.g.*, inconsistencies between variables and controller functions). This work is highly motivated by the weakly-typed nature of Javascript. The main issue is the erroneous understanding of the relationship between the JavaScript code and the API "*Document Object Model*", which is responsible to represent the hierarchy of HTML elements and their properties. The authors analyzed 300 bug reports to understand the issues specific for JavaScript applications (*e.g.*, their root cause and propagation). The static analysis is implemented in a tool called *AUREBESH*. This tool detected 15 bugs in 22 real-world MVC applications. They established four patterns to explain the root cause of these bugs. For example, one pattern, called *boolean assigned a string*, occurs when a developer assigning a string to a variable that expects a boolean value.

Regarding the **GUI design smells**, we first recall the characterization and detection

of object-oriented design smells to illustrate how they cannot be used in our work presented in Chapter 4. For instance, Fowler [Fow99] and Brown *et al.* [BMMM98] characterized multiple design smells, resp. anti-patterns, associated with refactoring operations. The detection of design smells mainly relies on the association of code metrics (*e.g.*, LOC or CC) to form heuristics. Closely related, Moha *et al.* propose *Decor*, a method to specify and then detect design smells [MGDLM10]. In this thesis, we characterized a new type of GUI design smell, called *Blob listener*, as a GUI listener that contain more than one GUI command (Section 4.3). From this definition, the aforementioned approach can be hardly used to detect *Blob listeners*. It would require considering the number of GUI commands per GUI listener as a code metric, and specify a rule based on this new metric to detect *Blob listeners*. If considering such a metric as a standard metrics is relevant given the increasingly interactivity of software systems, it is not possible yet. Moreover, our detection strategy aims at precisely locating each GUI commands that compose *Blob listeners*, not just detecting *Blob listeners*.

Sahin *et al.* propose an approach to generate design smell detection rules using a dedicated optimization technique [SKBD14]. The goal is to ease the definition of detection rules by automatically analyzing code instead of relying on a manual process. Closely related, Zanoni *et al.* propose an approach based on machine learning to detect design smells [ZFS15]. Khomh *et al.* propose goal question metric-based approach to detect anti-patterns [KVGS11]. Because of GUI specificities, the *Blob listener* detection rule we propose does not rely on standard code metrics. These approaches, however, may be applied on detected *Blob listeners* to see whether interesting detection rules or results are produced and to what extent they can be used to precise our approach. Several research work on design smell characterization and detection are domain-specific. For instance, Moha *et al.* propose a characterization and a detection process of service-oriented architecture anti-patterns [MPN$^+$12]. Garcia *et al.* propose an approach for identifying architectural design smells [GPEM09]. Similarly, this thesis aims at motivating that GUIs form another domain concerned by specific design smells that have to be characterized.

Unlike the aforementioned object-oriented design smells, less research work focuses on GUI design smells. We identified two research studies focus on design smells for GUIs. These studies, however, leverage static analysis to detect the presence of design smells in another type of GUI artifacts such as GUI dialog models [SCSS14] and GUI test scripts [CW12]. Also, the strategies of design smell detections may be based on metrics or existing OO design smells. Silva *et al.* propose an external bad smells detection in dialog models [SCSS14]. *External* by the authors means looking for bad smells from the running system instead of from source code analysis. Dialogue models describe the behaviour of a user interface such as GUI components and their relationship. The authors use the static analysis approach, tooled by *GUISurfer* [SSG$^+$10], to build such models. To detect the bad smells in the dialog models, the approach calculates two metrics: *Pagerank* and *Betweenness*. These two metrics are used to measure the user interface complexity. This complexity can reveal some bad smells such as *erroneous distributed complexity along the application behavior* or *misplaced central axes in the*

*interaction between users and the system*. The authors show how these metrics can be applied but no detail about the bad smells is provided.

Chen and Wang [CW12] identify 11 *bad smells in GUI test scripts* illustrated in Table 2.7. This choice is motivated by the fact that the structure of a GUI test script is similar to the source code (*e.g.*, statements) in some aspects. First, a GUI test script is a sequence of actions that will be executed over the GUI of a SUT. These actions are represented by GUI events or assertions, this concern the correctness of the SUT. Second, a method also contains a sequence of statements that will be executed when it is invoked. For example, the bad smell *long method* (*aka.* God Method [Fow99]), which is too large since it contains several statements, is analogous to the bad smell *long keyword*. This bad smell shows up in a keyword-driven test (KDT) script when one keyword contains several actions. These two bad smells are similar since they tackle the same problem but in different contexts: several *statements* in a method and several *actions* in a test script. Also, the authors propose 16 refactoring methods to remove automatically such bad smells. The bad smell *long keyword* can be removed by applying the method "*extract macro event*". This method removes several actions found into a *keyword* (or *macro event*) to create a new reduced one. In this thesis, we identify a new type of GUI design smell, *Blob listener*, that are also related to the design smell *long (God) Method*. *Blob listener* is a GUI particular instance of the *God Method*. However, we do not investigate the correlation between the *Blob listeners* and others GUI artifacts.

Table 2.7: Bad smells for GUI test scripts. Adapted from [CW12]

| **Bad Smell** | **Description** |
|---|---|
| *Unsuitable Naming* | A keyword, macro component, macro event, or primitive component is not properly named. |
| *Duplicated Actions* | Some duplicated actions appear in multiple places. |
| *Long Keyword* or *Long Macro Event* | A keyword (or macro event) contains too many actions. |
| *Long Parameter List* | The parameter list of a keyword or macro event is too long. |
| *Shotgun Surgery* | Multiple places need to be modified with a single change. |
| *Large Macro Component* | A single macro component contains too many primitive components, macro events, and macro components. |
| *Feature Envy* | A macro event uses another macro component's macro events or components excessively. |
| *Middle Man* | A macro component delegates all its tasks to another macro component. |
| *Lack of Macro Events* | A macro component does not have any macro events. |
| *Lack of Encapsulation* | Using primitive events directly instead of encapsulating actions into macro events. |
| *Inconsistent Hierarchy* | The macro component hierarchy is inconsistent with the structure of the GUI. |

One may note that metrics play a special role to the detection strategies of design smells. However, few studies focus on establishing **metrics** that are specific for GUIs.

Magel *et al.* [MA07] introduce five GUI structural metrics. We detail below three of them that deal with the complexity of GUIs.

- *Controls' Count (CC)/LOC* measures how much the code of a system is dedicated to implement a GUI. This metric is an adaptation of CC metric, which is based on the total number of controls' count that has an interface. A high CC value may indicate the system's complexity. However, this metric alone does not reflect the GUI complexity since some controls are easier to test than others.

- *The GUI tree depth* measures the depth of a GUI. A GUI structure is mapped to a tree model and then the depth of the tree is calculated based on its leaf deepest node. The authors believe that this metric can be useful to select more representative GUI test scenarios. For instance, the controls that have the same parent of the selected control may be excluded from a test scenario when it starts from the lowest level control. Such a reduction is based on heuristics that are not provided by the authors.

- *The structure of the tree* measures how much a GUI is complex. The complexity is based on the fact that if a tree has the most of GUI controls on the top, this tree is more complex. From a testing point of view, this kind of tree reflects more user choices or selections.

These metrics aim at measuring *how much effort is needed to test a particular GUI.* They do not indicate a problem for instance in the source code of a GUI.

Most of GUI research studies have used mature metrics such as CC to measure the complexity of a GUI code. Other metrics, however, may also be adapted to detect specific problems in the GUI code such as number of changes (per commit, class, or method) and cohesion metrics. For instance, the cohesion metric has been applied to indicate *fat interfaces* [RRP14], which are classes that handle several methods from different clients. In our work, this metric may be adapted to measure the cohesion of GUI controllers that are *Blob listeners*. We believe that several metrics can be used or adapted to help software engineers to identify problems in GUI code. In this thesis, we have used cyclomatic complexity as an initial step to investigate the code of GUI controllers to then establish a more elaborate and precise rule, *i.e.*, the number of GUI commands per GUI listener, to automatically detect *Blob listeners*.

In this section, we presented the GUI V&V techniques and the GUI defect classifications schemes to support them. The limitations of GUI schemes allowed us to explain the main drawbacks of current GUI testing techniques. On the one hand, GUI testing techniques aim at the reliability of standard GUIs. So, they fail in providing solutions to test advanced GUIs and their interactive features. By the other hand, GUI analysis approaches aim at the quality of GUIs. None of them have targeted design smells that may degrade the GUI code.

## 2.4 Conclusions

This chapter presented the state-of-the-art of testing GUI systems. GUIs play a vital role in designing interactive systems. The new trend in GUI design brings several specificities that impact on how to test GUIs. GUI testing has been applied to find GUI failures provoked by GUI defects. Several defect classification schemes have been proposed but few of them focus on GUIs. The gap is twofold. First, the proposed GUI defect classifications are not properly defined since they are based on a small set of standard widgets. Second, GUI schemes do not consider defects that arise from richer GUIs. For example, GUI errors that show up from the graphical nature of post-WIMP GUIs, their *ad hoc* widgets and complex interactions are not covered. Several kinds of faults specific to GUIs have been identified and described into a new GUI fault model, which will be presented in Chapter 3.

Different GUI V&V techniques are detailed. They vary in their purpose *w.r.t.* the GUI artifact used to provide automated solutions. Most of GUI approaches focus on automated GUI testing for standard GUIs. Also, MBT approaches that leverage dynamic analysis to build GUI test models have demonstrated their ability to detect crashes and regressions. Such approaches, however, require expressive GUI models to produce more effective test cases. Different kinds of UIDLs have thus been proposed to describe intrinsic components of GUI design, which are represented as GUI models. Such models have been used to derive GUI test models and thus support test case generation. Also, GUI V&V techniques inspect the source code to measure the code quality for instance detecting error-prone code that may result in GUI failures. Static analysis approaches have been applied for different purposes such as GUI testing, GUI maintenance and evolution. We identified few studies focus on GUI *design smells*. A new type of design smell specific for GUIs is identified and automatically detected by a novel GUI static analysis proposed in this thesis. We will present the details in Chapter 4.

# Part II

# Contributions

# Chapter 3

# GUI fault model

This chapter presents an original fault model for graphical user interfaces. This fault model is a GUI fault classification scheme structured at two levels: user interface faults and user interaction faults. For each fault we illustrate *which* GUI is affected and *how* it occurs as GUI failure (Section 3.1). We evaluate the coverage of our fault model through an empirical analysis: identifying and classifying several GUI failures from open-source bug repositories (Section 3.2). We also assess the ability of two GUI testing tools (GUITAR and Jubula) to find real GUI failures previously classified (Section 3.3).

The practical use of the GUI fault model is demonstrated by forging several mutants of a highly interactive open-source system (LaTeXDraw). These mutants implement the faults described in our fault model. We then conduct a third experiment to evaluate the ability of those GUI testing tools to detect these mutants (Section 3.4). We also discuss the reasons why several mutants are not killed. A precise analysis of the limits of GUI testing frameworks for testing advanced GUIs is presented in Section 3.6. We selected GUITAR, a standard GUI model-based testing framework, to study and explain the limitations of the current standard GUI testing approaches. We also present the concept of interaction-action-flow graph to tackle these limitations (Section 3.7). The benefits of this approach are demonstrated through two different use cases.

The contributions of this chapter have been published in [LBB15, LBBC15].

## 3.1 Fault Model

In this section we present an exhaustive GUI fault model. To build the fault model we first analyzed the state-of-the-art of HCI concepts (recall Chapter 2). We then analyzed real GUI bug reports (different than those used in Section 3.2) to assess and precise the fault model. We performed a round trip process between the analysis of HCI concepts and GUI bug reports until a stable fault model was obtained.

In Chapter 2, we present the definitions of fault, error, failure. Based on these definitions, we propose the following definitions of a *GUI* fault, error, and failure:

**Definition 3.1 (GUI Fault)** *GUI faults are differences between an incorrect and a correct behavior description of a GUI.*

**Definition 3.2 (GUI Error)** *A GUI error is an activation of a GUI fault that leads to an unexpected GUI state.*

**Definition 3.3 (GUI Failure)** *A GUI failure is a manifestation of an unexpected GUI state provoked by a GUI fault.*

A GUI fault can be introduced at different levels of a GUI software (*e.g.*, GUI code, GUI models). An illustration of a GUI fault is: a correct line of GUI code *vs* an incorrect line of GUI code. For example, a GUI fault can be activated when an *entry*, such as a value into an input widget, is not *handled correctly* by its GUI code. So, an unexpected GUI state is manifested (*e.g.*, a crash as a GUI failure) when a user clicks on a button after typing this entry.

Our fault model is divided into the following groups:

- *User interface faults* refer to faults that affect the structure and the behavior of graphical components of GUIs.

- *User interaction faults* refer to faults that affect the interaction process, *i.e.*, when a user interacts with a GUI.

### 3.1.1 User Interface Faults

GUIs are composed of widgets that can act as mediators to interact indirectly (*e.g.*, buttons in WIMP GUIs) or directly (direct manipulation principle in post-WIMP GUIs) with objects of the data model. In this section, we categorize the user interface faults, *i.e.*, faults related to the structure, the behavior, and the appearance of GUIs. We further break down user interface faults into two categories: the *GUI structure and aesthetics*, and the *data presentation* faults, as introduced below. Table 3.1 presents an overview of these faults and their potential failures.

#### 3.1.1.1 GUI Structure and Aesthetics Fault

This fault category corresponds to unexpected GUI designs. Since GUIs are composed of widgets laid out following a given order, the first fault is the *incorrect layout of widgets* (GSA1). Possible failures corresponding to this fault occur when GUI widgets follow an unexpected layout (*e.g.*, wrong size or position). The next fault concerns the *incorrect state of widgets* (GSA2). Widgets' behavior is dynamic and can be in different states such as visible, enabled, or selected. This fault occurs when the current state of a widget differs from the expected one. For example, a widget is unexpectedly visible. The following fault treats the *unexpected appearance of widgets* (GSA3). That concerns aesthetic aspects of widgets not bound to the data model, such as look-and-feels, fonts, icons, or misspellings.

Table 3.1: User Interface Faults

| Fault categories | ID | Faults | Possible failures |
|---|---|---|---|
| GUI Structure and Aesthetics | GSA1 | *Incorrect layout of widgets* (*e.g.*, alignment, dimension, orientation, depth) | The positions of 2 widgets are inverted.<br>A text is not fully visible since the size of text field is too small.<br>Rulers do not appear on the top of a drawing editor.<br>The vertical lines for visualizing the precise position of shapes in the drawing editor are not displayed. |
| | GSA2 | *Incorrect state of widgets* (*e.g.*, visible, activated, selected, focused, modal, editable, expandable) | Not possible to click on a button since it is not activated.<br>A window is not visible so that its widgets cannot be used.<br>Not possible to draw in the drawing area of a drawing editor since it is not activated. |
| | GSA3 | *Incorrect appearance of widgets* (*e.g.*, font, color, icon, label) | The icon of a button is not visible.<br>In a GUI of a power plant, the color reflecting the critical status of a pump is green instead of red. |
| Data Presentation | DT1 | *Incorrect data rendering* (*e.g.*, scaling factors, rotating, converting) | The size of a text is not scaled properly.<br>In a drawing editor, a dotted line is painted as a dashed one.<br>A rectangle is painted as an ellipse. |
| | DT2 | *Incorrect data properties* (*e.g.*, selectable, focused) | A web address in a text is not displayed as hyperlink. |
| | DT3 | *Incorrect data type or format* (*e.g.*, degree *vs* radian, float *vs* double) | The date is displayed with 5 digits (*e.g.*, dd/mm/y) instead of 6 digits (*e.g.*, dd/mm/yy).<br>A text field displays an angle in radian instead of in degree. |

#### 3.1.1.2 Data presentation

In many cases, widgets aim at editing and visualizing data. For example with WIMP GUIs, text fields or lists can display simple data to be edited by users. Post-WIMP GUIs share this same principle with the difference that the data representation is usually *ad hoc* and more complex. For example, the drawing area of a drawing editor paints shapes of the data model. Such a drawing area has been developed for the specific case of this editor. That permits to represent graphically in a single widget complex data (*e.g.*, shapes). In other cases, widgets aim at monitoring data only. This is notably the case for some GUIs in control commands of power plants where data are not edited but monitored by users. The definition of data representations is complex and error-prone. It thus requires adequate data presentation faults.

The first fault of this category is the *incorrect data rendering* (DT1). DT1 is pro-

voked when data is converted or scaled wrongly. Possible failures for this fault are manifested by unexpected data appearance (*e.g.*, wrong color, texture, opacity, shadow) or data layout (*e.g.*, wrong position, geometry). The second fault concerns *incorrect data properties* (DT2). Properties define specific visualization of data such as selectable or focused. A possible failure is a web address that is not displayed as a hyperlink. The last fault (DT3) occurs when an *incorrect data type or format* is displayed. For instance, an angle value is displayed in radian instead of in degree.

### 3.1.2 User Interaction Faults

In this section, we introduce the faults that treat user interactions. The proposed faults are based on the characteristics of WIMP and post-WIMP GUIs detailed in the previous section. For each fault we separated our analysis into two parts. One dedicated to WIMP interactions and another one to post-WIMP interactions. WIMP interactions refer to interactions performed on WIMP widgets. They are simple and composed of few events (click[1], key pressed, *etc.*).

Post-WIMP interactions refer to interactions performed on post-WIMP widgets. Such interactions are more complex since they can be *multimodal*, *i.e.*, involve multiple input devices (gesture, gyroscope, multi-touch screen); be *concurrent* (*e.g.*, in bi-manual interactions the two hands evolve in parallel); be *composed of numerous events* (*e.g.*, multimodal interactions may be composed of sequences of pressure, move, and voice events). Subsequently the direct manipulation principles, other particularities of post-WIMP interactions are that they aim at: being as natural as possible; providing users with the feeling of handling data directly (*e.g.*, shapes in drawing editors).

Table 3.2 summarizes the *user interaction* faults and some of their potential failures for both WIMP and post-WIMP interactions. These faults are detailed as follows.

#### 3.1.2.1 Interaction Behavior

Developing post-WIMP interactions is complex and error-prone. Indeed, as explained in Chapter 2, it may involve many sequences of events or require the fusion of several modalities such as voice and gesture. So, the first fault (IB1) occurs when the behavior of the developed *interactions does not work properly*. This fault mainly concerns post-WIMP widgets since WIMP widgets embed simple and hard-coded interactions. For instance, an event such as *pressure* can be missing in a bi-manual interaction. Another example is the incorrect synchronization between the voice and the gesture in a voice+gesture interaction. GUI toolkits, however, are progressively providing developers with predefined basic post-WIMP interactions. For instance, JavaFX introduced the swipe, zoom, and rotate gesture interactions[2].

[1] A click is one interaction composed of the event *mouse pressed* followed by the event *mouse released*. Its simple behavior has leaded to consider a click as an event itself.

[2] `https://docs.oracle.com/javafx/2/api/javafx/scene/input/GestureEvent.html`

Table 3.2: User Interaction Faults

| **Fault categories** | **ID** | **Faults** | **Possible failures** |
|---|---|---|---|
| Interaction Behavior | IB1 | Incorrect behavior of a *user interaction* | A bi-manual interaction developed for a specific purpose does not work properly.<br>The synchronization between the voice and the gesture does not work properly in a voice+gesture interaction. |
| Action | ACT1 | Incorrect *action results* | Translating a shape to a position $(x, y)$ translates it to the position $(-x, -y)$.<br>Setting the zoom level at 150%, sets it at 50%. |
| | ACT2 | *No action* executed | Clicking on a button has no effect.<br>Executing a DnD on a drawing area to draw a rectangle has no effect. |
| | ACT1 | *Incorrect action* executed | Clicking on the button *Save* shows the dialogue box used for loading.<br>Scaling a shape results in its rotation.<br>Performing a DnD to translate shapes results in their selection. |
| Reversibility | RVSB1 | Incorrect results of *undo or redo* operations | Clicking on the button *redo* does not re-apply the latest undone action as expected.<br>Pressing the keys *ctrl+z* does not revert the latest executed action as expected. |
| | RVSB2 | Reverting the *current interaction* in progress works incorrectly | Pressing the key "*Escape*" during a DnD does not abort this last.<br>Saying the word "*Stop*" does not stop the interaction in progress. |
| | RVSB3 | Reverting the *current action* in progress works incorrectly | Clicking on the button "*Cancel*" to stop the loading of the file previously selected does not work properly. |
| Feedback | FDBK1 | Feedback provided by widgets to reflect the *current state of an action in progress* works incorrectly | The progress bar that shows the loading progress of a file works incorrectly. |
| | FDBK2 | The temporary feedback provided all *along the execution of long interactions* is incorrect | Given a drawing editor, drawing a rectangle using a DnD interaction does not show the created rectangle during the DnD as expected. |

#### 3.1.2.2 Action

This category of faults groups faults that concern actions produced while interacting with the system. The first fault (ACT1) focuses on the *incorrect results of actions*. In this case the expected action is executed but its results are not correct. For instance with a drawing editor, a failure can be the translation of one shape to the given position $(-x, -y)$ while the position $(x, y)$ was expected. The root cause of this failure can be located in the action itself or in its settings. For instance, a first root cause of the previous failure can be the incorrect coding of the translation operation. A second root cause can be located in the settings of the translation action.

The second fault (ACT2) concerns the *absence of action* when interacting with the system. For instance, this fault can occur when an interaction, such as a keyboard shortcut, is not correctly bound to its widget.

The third fault (ACT3) consists of the *execution of wrong actions*. The root cause of this fault can be that the wrong action is bound to a widget at a given instant. For instance: clicking on the button *Save* shows the dialogue box used for loading; doing a DnD interaction on a drawing area selects shapes instead of translating them.

#### 3.1.2.3 Reversibility

This fault category groups three faults. The fault (RVSB1) concerns the *incorrect behavior of the undo/redo* operations. Undo and redo operations usually rely on WIMP widgets such as buttons and key shortcuts. These operations revert or re-execute actions *already terminated* and stored by the system. A possible failure is the incorrect reversion of the latest executed action when the key shortcut *ctrl+z* is used.

Contrary to WIMP interactions, that are mainly one-shot, many interactions last some time such as the DnD interaction. In such a case, users may be able to stop an interaction in progress. The second fault (RVSB2) thus consists of the *incorrect interruption of the current interaction* in progress. For instance, pressing the key "*Escape*" during a DnD does not stop this last as expected. This fault could have been classified as an interaction behavior fault. We decided to consider it as a reversibility fault since it concerns the ability to revert an ongoing interaction.

Once launched, actions may take time to be executed entirely. In this case such actions can be interrupted. The third fault (RVSB3) concerns the *incorrect interruption of an action* in progress. A possible failure concerns the file loading operation: clicking on the button "*Cancel*" to stop the loading of a file does not work properly.

#### 3.1.2.4 Feedback

Widgets are designed to provide immediate and continuous feedback to users while they interact with them. For instance, progress bars that show the loading progress of a file is a kind of feedback provided to users. The first fault of this category (FDBK1) concerns the *incorrect feedback provided by widgets*, where feedback reflects the current state of an action in progress. This fault focuses on actions that last in time and which progress should be monitored by users.

The second fault (FDBK2) focuses on *potentially long interactions* (*i.e.*, interactions that take a certain amount of time to be completed) which progress should be discernible by users. For instance with a drawing editor, when drawing a shape on the drawing area, the shape in creation should be visible so that the user knows the progression of her work. So, a possible failure is drawing a rectangle using a DnD interaction, that works correctly, does not show the created rectangle during the DnD as expected.

### 3.1.3 Discussion

The definition and the use of a fault model raise several questions we discuss about in this subsection.

**What are the benefits of the proposed GUI fault model?**

The benefits of our GUI fault model are twofold. First, a fault model is an exhaustive classification of faults for a specific concern [vBDD+91]. Providing a GUI fault model permits GUI developers and testers to have a precise idea of the different faults they must consider. As an illustration, Section 3.2 describes an empirical analysis we conducted to classify and discuss about GUI failures of open-source GUIs. Second, our GUI fault model allows developers of GUI testing tools to evaluate the efficiency of their tool in terms of bug detection power *w.r.t.* a GUI specific fault model. As detailed in Section 3.4, we created mutants of an existing GUI. Each mutant contains one GUI failure that corresponds to one GUI fault of our fault model. Developers of GUI testing tools can run their tools against these mutants for benchmarking purposes.

**Should usability have been a GUI fault?**

Answering this question requires the definition of a fault to be re-explained: a fault is a difference between the observed behavior description and the expected one. Usability issues consist of reporting that the current observed behavior of a specific part of a GUI lacks at being somehow usable. That does not mean the observed behavior differs from the behavior expected by test oracles. Instead, it usually means that the expected behavior has not been defined correctly regarding some usability criteria. That is why we do not consider usability as a GUI fault. This reasoning can be extended to other concerns such as performance.

**How to classify GUI failures into a fault model?**

A GUI failure is a perceivable manifestation of a GUI error. Classifying GUI failures thus requires to have identified the root cause (*i.e.*, GUI fault) of the failure. So, classifying GUI failures can be done by experts of the GUI under test. These experts need sufficient information, such as patches, logs, or stack traces, to identify whether the root cause of a failure is a GUI fault to then classify it. For example, given a failure manifested through the GUI and caused by a precondition violation. In this case, such a failure is not classified into the GUI fault model. Similarly, classifying correctly a GUI failure also requires to qualify the involved widgets (*e.g.*, standard or *ad hoc*) as well as the interaction (*e.g.*, mono-event or multiple-event interaction).

**How to classify failures stemming from other failures?**

For instance, the incorrect results of the execution of an action (action fault) let a widget not visible as expected (GUI structure fault). In such cases, only the first failure must be considered since it puts the GUI in an unexpected and possibly unstable state. Besides, the appearance of a GUI error depends on the previous actions and interactions successfully executed. Typical examples are the undo and redo actions. A redo action can be executed only if an action has been previously performed. Furthermore, the success of a redo action may depend on the previous executed actions. We considered this point during the creation of mutants (as detailed in Section 3.4) to provide failures that appear both with and without previous actions.

In the next three sections, we present empirical studies that assess the relevance of our GUI fault model when applied to real situations.

## 3.2 Relevance of the Fault Model: an empirical analysis

In this section the proposed GUI fault model is evaluated. Our evaluation has been conducted by an empirical analysis to assess the relevance of the model *w.r.t.* faults currently observed in existing GUIs. The goal is to state whether our GUI fault model is relevant against failures found in real GUIs.

### 3.2.1 Introduction

To assess the proposed fault model, we analyzed bug reports of five popular open-source software systems: Sweet Home 3D, File-roller, JabRef, Inkscape, and Firefox Android. These systems implement various kinds of widgets, interactions, and encompass different platforms (desktop and mobile). Their GUIs cover the main following features: *indirect and direct* manipulation; *several input devices* (*e.g.*, mouse, keyboard, touch); *ad hoc widgets* such as canvas; *discrete data manipulation* (*e.g.*, vector-based graphics); and *undo/redo* actions.

### 3.2.2 Experimental Protocol

Bug reports have been analyzed manually from the researcher/tester perspective by looking only at data available in the failures report (*i.e.*, black box analysis). To focus on detailed and commented bug reports that concern GUI failures, the selection has been driven by the following rules. Only closed, fixed, and in progress bug reports were selected. The following *search string* has been also used to reduce the resulting sample:

(*interface OR "user interface" OR "graphical user interface" OR "graphical interface" OR GUI OR UI OR layout OR design OR graphic OR interaction OR "user interaction" OR interact OR action OR feedback OR revert OR reversible OR undo OR redo OR abort OR stop OR cancel*).

Each report has been then manually analyzed to state whether it is a GUI failure. Also, selected bug reports have to provide explanations about the root cause of the failure such as a patch or comments. This step is crucial to be able to categorize the failures using our GUI fault model considering their root cause. We also discarded failures identified as non-reproducible, duplicated, usability, or user misunderstanding. From this selection we kept 279 bug reports (in total for the five systems) that describe one GUI failure each. The following sub-sections discuss about these failures and the classification process.

### 3.2.3 Classification and Analysis

All the 279 failures have been successfully classified into our fault model. Figure 3.1 gives an overview of the selected and classified bug reports. These failures were classified into the *Action* (119 failures, 43%), *GUI Structure and Aesthetics* (75 failures, 27%), *Data Presentation* (39 failures, 14%), *Reversibility* (31 failures, 11%), *Interaction behavior* (12 failures, 4%), and *Feedback* (3 failures, 1%) fault categories. Most of the failures classified into *GUI Structure and Aesthetics* concern the *incorrect layout of widgets* (51%). Likewise, most of the failures in the *Action* category refer to *incorrect action results* (75%).

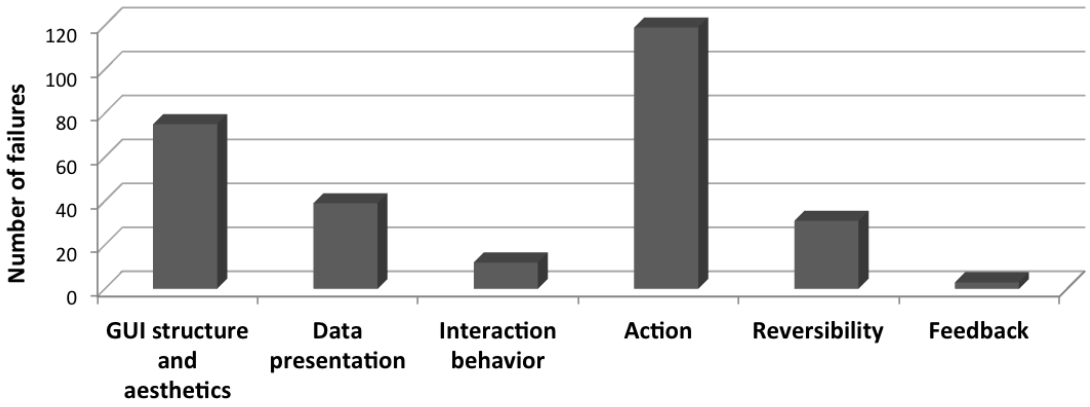


Figure 3.1: Classification of the 279 bug reports using the GUI fault model

Table 3.3 shows the distribution of the 279 analyzed GUI failures per software and category (user interface or user interaction). These results point out that the systems *Sweet Home 3D* and *Firefox Android* seem to be more affected by user interface failures. Most of these failures concern the *GUI structure and aesthetics* fault. That can be explained by the complex and *ad hoc* GUI structure of these systems. *File Roller* and *JabRef* GUIs include widgets with coarse-grained properties (*i.e.*, simple input value such as number or text). Most of their failures concern WIMP interactions classified into the *action* category. In contrast, *Inkscape* presented more failures classified as post-WIMP. Indeed, Inkscape, a vector graphics software, mainly relies on its drawing

area that provides users with different post-WIMP interactions. These failures have been categorized mainly into *interaction behavior*, *action*, and *reversibility*.

Table 3.3: Distribution of analyzed failures per software

| GUI Software | Analyzed failures | User interface failures | User interaction failures | Bug Repositories link |
|---|---|---|---|---|
| Sweet Home 3D | 33 | 55% | 45% | http://sourceforge.net/p/sweethome3d/bugs/ |
| File-roller | 32 | 28% | 72% | https://bugzilla.gnome.org/query.cgi |
| JabRef | 84 | 42% | 58% | http://sourceforge.net/p/jabref/bugs/ |
| Inkscape | 82 | 28% | 72% | https://bugs.launchpad.net/inkscape/ |
| Firefox Android | 48 | 60% | 40% | https://bugzilla.mozilla.org/ |

As depicted by Figure 3.2, 41% of these 279 GUI failures are originated by faults classified into the user interface category and 59% into the user interaction category. Most of user interaction failures have been classified into the *incorrect action results* (54%). This plot also highlights that only 25% of the analyzed user interface failures and 18% of the user interaction ones have been classified as post-WIMP. We comment these results in the following subsection.

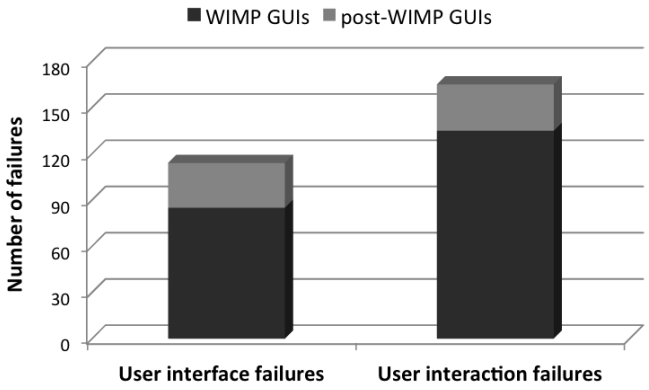


Figure 3.2: Manifestation of failures in the user interface and interaction levels

### 3.2.4 Discussion

The empirical results must be balanced with the fact that user interactions are less tangible than user interfaces. So, users may report more GUI failures when they can perceive failures graphically (an issue in the layout of a GUI or in the result of an action visible through a GUI). Users, however, may have difficulties to detect a failure in an interaction itself while interacting with the GUI. That may explain the low number of failures (4%) classified into *Interaction Behavior*. Another explanation may be the primary use of WIMP widgets, relying on simple interactions.

In our analysis, many failures that could be related to *Feedback* were discarded since they concerned enhancements or usability issues, which are out of the scope of a GUI fault model as discussed previously. For instance, GUI failures that concern the

lack of haptic feedback in Firefox Android were discarded. So, few faults (1%) were classified into this category. Another explanation may be the difficulty for users to identify feedback issues as real failures that should be reported.

We observed that some reported GUI failures are false positives regarding the *fault localization*: if the report does not have enough information about the root cause of a failure (*e.g.*, patch or exception log), a GUI failure can be classified in a wrong fault category. For example, when moving a shape using a DnD does not move it. At a first glance, the root cause of this failure can be associated to an incorrect behavior of the DnD. So, this failure can be categorized into the interaction behavior. However, after analyzing the root cause this failure refers to an action failure since the DnD works properly, but no action is linked to this interaction.

Likewise, the failures related to *Reversibility* and *Feedback* were easily identified through the steps to reproduce them. For example in JabRef, "*pressing the button "Undo" will clear all the text in the field, but then pressing the button "Redo" will not recover the text*". Furthermore, some systems do not revert interactions step by step but entirely. This can imply a failure from a user's point view, but sometimes it is considered as an invalid failure (*e.g.*, requirements vs. usability issues) by developers. In *JabRef*, the undo/redo actions did not revert discrete operations. For example, pressing the button "Undo" clears all texts typed into different text fields instead of clearing only one field each time the button "undo" is pressed.

Another important point concerns the WIMP vs. post-WIMP GUIs faults. We classified more failures involving WIMP than post-WIMP widgets. A possible explanation is that, despite the increasing interactivity of GUIs, the analyzed GUIs still rely more on WIMP widgets and interactions. Moreover, users now master the behavior of WIMP widgets so that they can easily identify when they provoke failures. It may not be the case with *ad hoc* and post-WIMP widgets.

## 3.3 Are GUI Testing Tools Able to Detect Classified Failures? An Empirical Study

This section provides an empirical study of two GUI testing tools: GUITAR and Jubula[3]. To demonstrate the current limitations of GUI testing tools in testing real GUIs, we applied those tools to detect the failures previously classified into our GUI fault model.

### 3.3.1 GUITAR and Jubula

GUITAR is one of the most widespread academic GUI testing tools. It extracts the GUI structure by reverse engineering. This structure is transformed into a GUI Event Flow Graph (EFG), where each node represents a widget event. Based on this EFG, test cases are generated and executed automatically over the SUT. We used the plugin for

[3] `http://www.eclipse.org/jubula`

Java Swing (*i.e.*, JFC GUITAR version 1.1.1)[4]. In GUITAR, each test case is composed by a sequence of widget events. The generation of test cases can be parameterized with the size of that sequence (*i.e.*, test case length).

Jubula is a semi-automated GUI testing tool that leverages pre-defined libraries to create test cases. These libraries contain modules that can be reused to generate manually test sequences. The modules encompass actions (*e.g.*, check, select) and interactions (*e.g.*, click, drag and drop) over different GUI toolkits (*e.g.*, swing, SWT, RCP, mobile). We have reused the library dedicated to Java Swing (Jubula version 7.2) to write the test cases presented in the next experiments. This library contains actions to test only standard widgets such as dragging a column/row of a table by passing an index. To test *ad hoc* widgets (*e.g.*, canvas), we made a workaround by mapping actions directly to these widgets. For example, to draw a shape on canvas we need to specify the exact position (*e.g.*, drag and drop coordinates) where the interaction should be executed.

### 3.3.2 Experiment

We selected JabRef[5], a software to manage bibliographic references. JabRef is written in Java which allows us to apply both GUITAR and Jubula. For each fault described in our GUI fault model, we selected one reported failure. To reproduce each failure, we downloaded the corresponding faulty version of JabRef. We used the exact test sequence (*i.e.*, number of actions) to reproduce a failure. In GUITAR, all test cases were generated automatically over a faulty version. In Jubula, each test case was created manually to detect one failure. Also, their test sequences are extracted by analyzing failure reports (*e.g.*, steps to reproduce a failure) and reusing Jubula's libraries. Then, GUITAR and Jubula run all their test cases automatically for checking whether the selected failure is found.

### 3.3.3 Results and Discussion

Table 3.4 summarizes the detection of the JabRef GUI failures by GUITAR and Jubula. These failures cover 11 out of the 15 faults described in our fault model. The remaining four faults were not covered for two reasons:

1. No failure was classified for that fault; or
2. A failure was classified, but we could not reproduce it - only occurred in a specific environment (*e.g.*, Operating System) or given a certain input (*e.g.*, a particular database in JabRef).

The reported failures in JabRef are mostly related to WIMP widgets, so we would expect GUITAR and Jubula to detect them, but it was not the case. For instance, failure #1 reports an incorrect display of buttons' label; its root cause is the incorrect size of a widget positioned to the left of them. Thus, this failure does not affect the

[4] `http://sourceforge.net/apps/mediawiki/guitar/`
[5] `http://jabref.sourceforge.net/`

Table 3.4: JabRef failures detected by GUITAR and Jubula

| ID fault | ID failure | Bug repository link | GUITAR | Jubula |
|---|---|---|---|---|
| GSA1 | #1 | http://sourceforge.net/p/jabref/bugs/160/ | ✗ | ✗ |
| GSA2 | #2 | http://sourceforge.net/p/jabref/bugs/514/ | ✗ | ✓ |
| GSA3 | #3 | http://sourceforge.net/p/jabref/bugs/166/ | ✗ | ✓ |
| DT1 | #4 | http://sourceforge.net/p/jabref/bugs/716/ | ✗ | ✓ |
| DT2 | #5 | - | | |
| DT3 | #6 | http://sourceforge.net/p/jabref/bugs/575/ | ✗ | ✓ |
| IB1 | #7 | - | | |
| ACT1 | #8 | http://sourceforge.net/p/jabref/bugs/495/ | ✓ | ✓ |
| ACT2 | #9 | http://sourceforge.net/p/jabref/bugs/536/ | ✓ | ✓ |
| ACT3 | #10 | http://sourceforge.net/p/jabref/bugs/809/ | ✗ | ✗ |
| RVSB1 | #11 | http://sourceforge.net/p/jabref/bugs/560/ | ✗ | ✓ |
| RVSB2 | #12 | - | | |
| RVSB3 | #13 | http://sourceforge.net/p/jabref/bugs/458/ | ✓ | ✓ |
| FDBK1 | #14 | http://sourceforge.net/p/jabref/bugs/52/ | ✗ | ✓ |
| FDBK2 | #15 | - | | |

values of internal properties (*e.g.*, text, event handlers) of those buttons. In GUITAR, checking the properties of that widget did not reveal this failure since the expected and actual values of its size property (*e.g.*, width) remained the same. In Jubula, the concerned widget cannot be mapped to test cases execution and thus cannot be tested.

Failures #2 and #3 refer to an incorrect menu path and a misspelling, respectively. Both failures were detected by Jubula. However, these failures were not found by GUITAR. Indeed, GUITAR does reverse engineering of an existing GUI to produce tests. If this GUI is faulty, GUITAR will produce tests that will consider these failures as the correct behavior.

Failures #8 and #13 that lead to a crash of the GUI were found by both GUITAR and Jubula. However, failures #4, #6, #10, and #11 that affect the data model were not detected by GUITAR for two reasons. First, GUITAR does not test the table entries in JabRef since they represent the data model. To do this, we need to extend GUITAR to interact with them. Second, the test cases successfully passed, but a failure has been revealed. That means, the events are fired properly (*e.g.*, no exception) and GUI properties are the "expected" ones. For example, a text property of a status bar contains the value: *"Redo: change field"*, when this action was actually not redone. Similarly, failure #10 was not detected by Jubula. This failure reports an unexpected auto-completion when the action "save" is triggered by shortcuts. We reproduced this failure manually but the test case was successfully replayed by Jubula. The input text via keyboard was typed and saved automatically without any interference of the auto-completion feature.

Another point is the accuracy of test cases generated manually in Jubula. Detecting failure #6 depends on how the test case is written. For example, adding a field that contains LaTeX commands (*e.g.*, 100\%), and then checking its output in a preview window should not contain any command (*e.g.*, 100%). So, we can write a test case to test the outputs in the preview window only looking for commands (*e.g.*, *SelectPattern[%, equals] in ComponentText[preview]*). Or, write a test case to check whether an

entire text matches to the expected one (*e.g.*, *CheckText[100%, equals] in ComponentText[preview]*). However, the last test case will fail since a text from preview window in JabRef is shown internally as HTML and, in Jubula, the action's parameters cannot be specified in that format.

Our experiment does not aim at comparing both tools since GUITAR is a fully automated tool contrary to Jubula. However, the results of this study highlight the current limitations of GUI testing tools. GUITAR and Jubula currently mainly work for detecting failures that affect properties of standard widgets. Moreover, GUITAR does GUI regression testing: it considers a given GUI as the reference one from which tests will be produced. If this GUI is faulty, GUITAR will produce tests that will consider these failures as the correct behavior. A possible solution to overcome this issue is to base the test process on the specifications (requirements, *etc.*) of the GUI.

## 3.4 Forging faulty GUIs for benchmarking

In this section, we evaluate the usefulness of our fault model by applying it on a highly interactive open-source software system: LaTeXDraw[6]. We created mutants of this system corresponding to the different faults of the model. The main goal of these mutants is to provide GUI testers with benchmark tools to evaluate the ability of GUI testing tools to detect GUI failures. As an illustration of the practical use of these mutants, we executed two GUI testing tools against the mutants of the system. Thanks to that we caught a glimpse of their ability to cover our proposed fault model. The goal of this experiment is to answer the research question: *what are the benefits of this fault model for GUI testing?*

### 3.4.1 Presentation of LaTeXDraw

LaTeXDraw is an open-source drawing editor for LaTeX that we selected as an illustration of interactive systems relying on more natural but complex interactions and widgets. We have selected LaTeXDraw to perform our experiments since it is a highly interactive system written in Java and Scala (dedicated to the creation of drawings for LaTeX), and its GUI mixes both standard and *ad hoc* widgets. Also, LaTeXDraw is released under an open-source license (GPL2) so that it can be freely used by the testing community. Figure 3.3 shows LaTeXDraw's GUI composed of both standard widgets grouped in two toolbars, and an *ad hoc* widget corresponding to the drawing area of the editor. This drawing area is special compared to standard widgets and is representative of other *ad hoc* widgets because of the three following points:

1. Data are dynamically presented in the widget in an *ad hoc* way. In our case, data is graphically represented in 2D.

2. Data presentations can be supplemented with widgets to interact directly with the

[6] `http://sourceforge.net/projects/latexdraw/`

underlying data. For example with Figure 3.3, the shape "Circle" is surrounded by eight scaling handlers and one rotation handler.

3. *Ad hoc* interactions are provided to interact with the data representations and their associated widgets. These interactions are usually more complex than standard widget's interactions. For instance, the scaling and rotation handlers in LaTeXDraw can be used with a drag-and-drop (DnD) interaction. With a multi-touch screen, one can zoom on shapes using two fingers (a bi-manual interaction) as it is typically more and more the case with mobile phones and tablets.

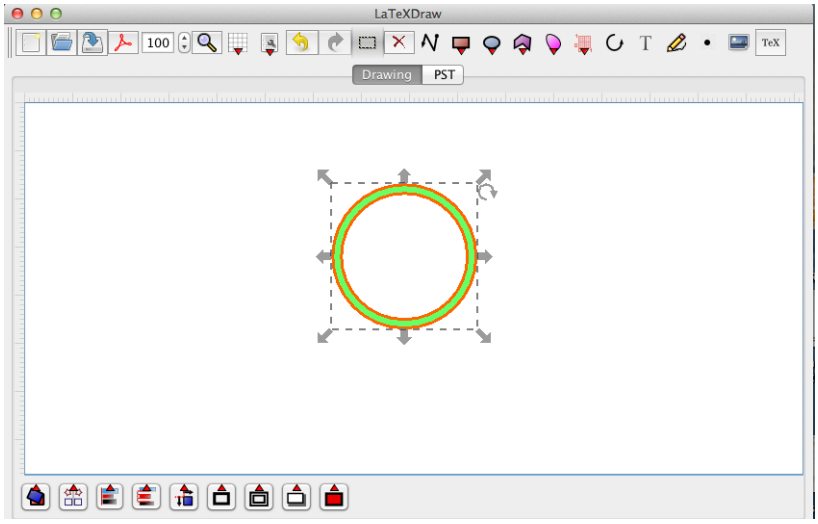


Figure 3.3: Screen-shot of the LaTeXDraw's GUI

**Failures to find in LaTeXDraw**

LaTeXDraw's GUI is composed of several standard widgets that can be tested with current GUI testing frameworks to find crashes. However, one may want to detect that executing one interaction on a GUI has the expected results. For instance, if a user performs a bi-manual interaction on the drawing area, the expected results are either a zoom action if no shape are targeted by the interaction; or a scale action applied on the targeted shape. The zoom and the scale levels are both computed from information provided by the interaction. One may want to validate that the correct action has been performed and that the execution of the action had the expected results (*e.g.*, no crash or the zoom level is correct). This kind of faults consists in validating that the specifications have been implemented correctly.

### 3.4.2 Mutants Generation

As highlighted by Zhu *et al.*, *"software testing is often aimed at detecting faults in software. A way to measure how well this objective has been achieved is to plant some*

*artificial faults into the program and check if they are detected by the test. A program with a planted fault is called a mutant of the original program"* [ZHM97]. Following this principle, we planted 65 faults in LaTeXDraw, a highly interactive open-source software system.

We created 65 mutants corresponding to the different faults of our proposed fault model. All these mutants and the original version are freely available[7]. Each mutant is documented to detail its planted fault and the oracle permitting to find it[7]. Multiple mutants have been created from each fault by: using WIMP (22 mutants) or post-WIMP (43 mutants) widgets to kill the mutants; varying the test case length (*i.e.*, the number of actions required to provoke the failure). Each action (*e.g.*, select a shape) requires a minimal number of events (*e.g.*, in LaTeXDraw a DnD requires at least three events: press/move/release) to be executed. Table 3.5 illustrates the attributes and their values for a GUI mutant[8] planted in LaTeXDraw concerning the *Action* category.

Table 3.5: Example of a GUI mutant in LaTeXDraw

| **Mutant attribute** | **Value** |
|---|---|
| Fault category | Action |
| Fault | No action is executed |
| Failure description | Any shape available in a toolbar is drawn. |
| WIMP/post-WIMP | post-WIMP |
| *Ad hoc* widgets involved | yes |
| Test case length | 2 (Select Shape, Add Shape) |
| Interactions | DnD, button click |
| Minimal number of events to perform | 4 (press button, press canvas, move canvas, release canvas) |
| Mutated files | The mutated file is package net.sf.latexdraw.ui.UIBuilder<br>The mutated code is the setEventableToInstruments() |
| Steps to produce the failure | 1. Select any shape style in the tool bar;<br>2. Drag and drop to draw a shape on canvas.<br>ORACLE -> The shape is not drawn. |

Tables 3.6 and 3.7 summarize the number of forged mutants and the minimal and maximal test case length for user interface and user interaction faults, respectively. For example, a length 0..2 means there exists at least one mutant requiring a minimum of 0 action or a maximum of 2 actions. However, the fault RVSB3 is currently not covered by the LaTeXDraw mutants. Similarly, some planted mutants only rely on post-WIMP interactions or widgets (*e.g.*, IB1, DT1).

[7] `https://github.com/arnobl/latexdraw-mutants`
[8] `https://github.com/arnobl/latexdraw-mutants/tree/master/GUImutants/mutant35`

Table 3.6: Mutants planted according to the user interface faults

| **ID** | GSA1 | GSA2 | GSA3 | DT1 | DT2 | DT3 |
|---|---|---|---|---|---|---|
| **#Mutants** | 3 | 7 | 4 | 2 | 1 | 1 |
| **#Length** | 0..2 | 0..4 | 0..4 | 4 | 2 | 4 |

Table 3.7: Mutants planted according to the user interfaction faults

| **ID** | IB1 | ACT1 | ACT2 | ACT3 | RVSB1 | RVSB2 | RVSB3 | FDBK1 | FDBK2 |
|---|---|---|---|---|---|---|---|---|---|
| **#Mutants** | 1 | 8 | 16 | 5 | 9 | 2 | - | 3 | 3 |
| **#Length** | 4 | 1..7 | 1..8 | 4..5 | 2..8 | 2 | - | 2..5 | 3..4 |

### 3.4.3 How GUI testing tools kill our GUI mutants: a first experiment

We applied the GUI testing tools GUITAR and Jubula on the mutants to evaluate their ability to kill them. Our goal is not to provide benchmarks against these tools but rather highlight the current challenges in testing interactive systems not considered yet (*e.g.*, post-WIMP interactions). GUITAR test cases have been generated automatically while Jubula ones have been written manually.

Considering the mutants planted at the user interface level, Jubula and GUITAR tests killed the mutants that involve checking standard widget properties, such as layout (*e.g.*, width, height) and state (*e.g.*, enable, focusable). Also, it is possible to test simple data (*e.g.*, string values on text fields) on those widgets. However, most of the mutants that concern the *ad hoc* widgets were alive. Notably, when test cases involve testing complex data from the data model. For example, it is not possible to compare the results of the actual shape on canvas against the expected one. Even if some shape properties (*e.g.*, rotation angle) are presented on standard widgets (*e.g.*, spinner), GUITAR and Jubula cannot state whether the current values in these widgets match the expected shape rotation on the canvas.

Likewise, our GUITAR and Jubula tests cannot kill most of the user interaction mutants that result on a wrong presentation of shapes. In particular, when we tested mutants planted into the *Reversibility* or *Feedback* categories. For example, testing undo/redo operations in LaTeXDraw should compare all states to manipulate a shape on canvas. Moreover, the tests verdict on Jubula passed even though interactions are defined incorrectly (*e.g.*, mouse cursor does not follow a DnD) or actions cannot be executed (*e.g.*, a button is disabled). In GUITAR, the generated test cases do not cover properly actions having dependencies. For example, the action "*Delete*" in LaTeXDraw requires first selecting a shape on canvas. However, no test sequence that contains "*Select Shape*" before "*Delete Shape*" was generated. Thus, some mutants could not be killed.

Table 3.8 gives an overview of the number of mutants killed by GUITAR and Jubula. The results show that both tools are not able to kill all mutants because of the four following reasons:

1. *Testing LaTeXDraw with GUITAR and Jubula is limited to the test of the standard*

*Swing widgets*. In Jubula, the test cases are only written according to libraries available in the Swing toolkit. In GUITAR, the basic package for Java Swing GUIs only covers standard widgets and mono-events (*e.g.*, a click on a button).

2. *Configure or customize a GUI testing tool to test post-WIMP widgets is not a trivial task*. For example, each sequence of a test case in Jubula needs to be mapped for the corresponding GUI widget manually. Also, GUITAR needs to be extended to generate test cases for *ad hoc* widgets (*e.g.*, canvas) as well their interactions (*e.g.*, multi-modal interactions).

3. *Testing post-WIMP widgets requires a long test case sequence*. In LaTeXDraw, a sequence to test interactions over these widgets is composed of at least two actions. That sequence is longer when we have to detect failures into undo/redo operations.

4. *It is not possible to give a test verdict for complex data*. The oracle provided by the 2 GUI testing tools do not know the internal behavior of *ad hoc* widgets, their interaction features and data presentation.

These results answer the research question by highlighting the benefits of our fault model for measuring the ability of GUI testing tools in finding GUI failures.

Table 3.8: Mutants killed by GUITAR and Jubula

| | **GUITAR** | | **JUBULA** | |
|---|---|---|---|---|
| **ID** | WIMP | post-WIMP | WIMP | post-WIMP |
| GSA1 | 2 | 0 | 2 | 0 |
| GSA2 | 5 | 0 | 6 | 1 |
| GSA3 | 3 | 0 | 3 | 0 |
| DT1 | - | 0 | - | 0 |
| DT2 | - | 0 | - | 0 |
| DT3 | - | 0 | - | 1 |
| IB1 | - | 0 | - | 0 |
| ACT1 | 0 | 0 | 0 | 1 |
| ACT2 | 3 | 0 | 3 | 0 |
| ACT3 | 2 | 0 | 2 | 0 |
| RVSB1 | 2 | 0 | 2 | 0 |
| RVSB2 | - | 0 | - | 0 |
| RVSB3 | - | - | - | - |
| FDBK1 | 1 | 0 | 1 | 0 |
| FDBK2 | - | 0 | - | 0 |

## 3.5 Threats to Validity

Regarding the conducted empirical studies, we identified the two following threats to validity. The first one concerns the scope of the proposed fault model since we evaluated it empirically on a small number (5) of interactive systems. To limit this threat, we selected interactive systems that cover different aspects of the HCI concepts we detailed in Chapter 2. The second threat we identified concerns the subjectivity observed in bug

reports to describe failures. To deal with this, we based the classification on the bug report artifacts (patches, logs, *etc.*) to identify the root cause of the reported failures.

## 3.6 Current limitations in testing advanced GUIs

Using LaTeXDraw as an illustrative example, we explain in this section the limitations of the current approaches for testing advanced GUIs (*i.e.*, post-WIMP GUIs) and thus for detecting failures such like those detailed in Section 3.4.1.

### 3.6.1 Studying failures found by current GUI testing frameworks

GUITAR has demonstrated its ability to find failures in standard GUIs [NRBM14] (*i.e.*, WIMP GUIs). It follows the standard GUI testing process depicted by Figure 2.6 in Chapter 2 and can be thus studied to highlight and explain the limitations of the current GUI testing approaches for testing advanced GUIs.

GUITAR developers provide information on 95 major failures detected by this tool during the last years in open-source interactive systems[9]. An analysis of these failures shows that:

1. All of them have been provoked by the use of *standard widgets* with *mono-event interactions*, mainly buttons and text fields;

2. All of them are *crashes*.

While several of the tested interactive systems provide advanced interactive features, all the reported failures are related to standard widgets. For instance, ArgoUML[10] is a modeling tool having a drawing area for sketching diagrams similarly to LaTeXDraw. None of the nine failures found by GUITAR on ArgoUML has been detected by interacting with this drawing area.

We applied GUITAR on LaTeXDraw to evaluate how it manages the mix of both standard widgets and *ad hoc* ones, *i.e.*, the drawing area and its content. If standard widgets were successfully tested, no test script interacted with the drawing area. In the next section, we identify the reasons of this limit and explain what is mandatory for resolving it.

### 3.6.2 Limits of the current GUI testing frameworks

The main differences between standard widgets and *ad hoc* ones are: one can interact with standard widgets using a single mono-event interaction while *ad hoc* ones provide multiple and multi-event interactions; *ad hoc* widgets can contain other widgets and data representations (*e.g.*, shapes or the handlers to scale shapes in the drawing area). As previously explained, four elements are involved in the typical GUI testing process: the GUI oracle; the test model; the language used to build GUI models; the model

[9] https://spreadsheets0.google.com/spreadsheet/pub?key=0Ah-FLr0Egz9zdGtINk54ZDlDbWdaNWdOSzR0clB4ZkE

[10] http://argouml.tigris.org/

creation process. In this subsection we explain how current GUI and test models hinder the ability to test advanced GUIs.

**Current GUI models are not expressive enough**. Languages used to build GUI models, called UIDLs, are a corner-stone in the testing process. Their expressiveness has a direct impact on the concepts that can compose a GUI test model (*e.g.*, an EFG). That has, therefore, an impact on the ability of generated GUI test cases to detect various GUI failures. For instance, GUITAR uses its own UIDL that captures GUI structures (the widgets that compose a GUI and their layout). However, in the current trend of developing highly interactive GUIs that use *ad hoc* widgets, current UIDLs used to test GUIs are no longer expressive enough as pointed out below.

1. *UIDLs currently used for GUIs testing describe the widgets but not how to interact with them.* The reason behind this choice is that current GUI testing frameworks test standard widgets, which behavior is the same in many GUI platforms. For instance, buttons work by pressing on it using a pointing device on all GUI platforms. This choice is no more adapted for advanced GUIs that rely more and more on *ad hoc* widgets and interactions. Indeed, the behavior of these tests has been developed specially for a GUI and is thus not standard. As depicted in Figure 3.4, GUITAR embedds the definition of how to interact with widgets directly in the Java code of the framework. Test scripts notify the framework of the widgets to use on the SUT. The framework uses its widgets definitions to interact with them. So, supporting a new widget implies extending the framework. Even in this case, if users can interact with a widget using different interactions the framework randomly selects one of the possible interactions. Thus, the choice of the interaction to use must be clearly specified in GUI models. That will permit to generate a test model (*e.g.*, an EFG) that can explore all the possible interactions instead of a single one. UIDLs must be expressive enough for expressing such interactions in GUI models.

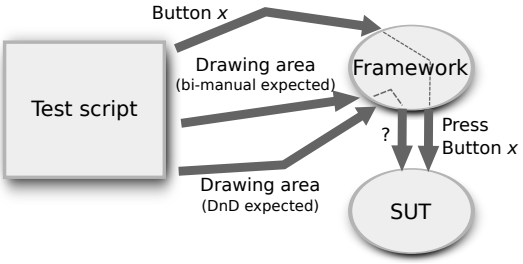


Figure 3.4: Representation of how interactions are currently managed and the current limit

2. *Current UIDLs do not support multi-event interactions.* The current trend in GUI design is to rely more on natural but complex interactions (*i.e.*, multi-event interactions such as the DnD and the bi-manual interactions) rather than on

mono-event ones (*e.g.*, button pressure). Moreover, such interactions can be *ad hoc*. Testing such interactions implies that UIDLs must be able to specify them. Following the previous point on describing interactions in GUI models, the main benefit here would be the ability to generate from GUI models test models leveraging the definition of these interactions.

3. *Current UIDLs do not describe the result of the use of one widget on the system.* Users interact with a GUI in a given purpose. The action resulting from the interaction of a user with a GUI should be described in GUI models. Similarly, the dependencies between actions and the fact that some actions can be undone and redone should be specified as well. For instance, an action *paste* can be performed only if an action *copy* or *cut* has been already performed. That would help the creation of GUI oracles able to state whether the result of one interaction had the expected results. For instance with LaTeXDraw, a GUI model could specify that executing a DnD on the drawing area can move a shape at a specific position if a shape is targeted by the DnD. From this definition, a GUI oracle could check that on a DnD the shape has been moved at the expected position. Another benefit would be the ability to reduce the number of the generated test cases by leveraging the dependencies between actions. Such work has been already proposed by Cohen *et al.* [CHM12] and must be capitalized in GUI models.

**Current EFGs mix both interaction, widget, and action under the term *event*.** Test models, *i.e.*, EFGs in our case, are graphs of all the possible *events* that can be produced by interacting with a GUI. An event is both the widget used and its underlying interaction. The name of the event usually gives an indication regarding the action produced when interacting with the widget. For instance, Cohen *et al.* describe a test model example is composed of the events *Draw*, *Paste*, *Copy*, *Cut*, *etc.* corresponding to both the menu items that produce these events and their interaction, here *Menu Pressed* [CHM12]. The name of each event describes the action resulting from the use of the widget. However, this mix of concepts may hamper the testing process of advanced GUIs as explained as follows.

1. *Actions must be reified in EFGs.* Currently, GUI oracles detect crashes or regressions and are not supplemented with information coming from EFGs. However, testing advanced GUIs requires the detection of other kinds of failures as explained in Section 3.4.1. In this chapter, we empirically identified and classified various kinds of GUI failures that can affect both WIMP and post-WIMP GUIs (see Section 3.1). For instance with LaTeXDraw, a GUI oracle must be able to state whether a shape has been correctly moved. EFGs should contain information about the actions defined in GUI models in order to provide GUI oracles with information mandatory for stating on test verdicts.

2. *Interactions and widgets must be clearly separated in EFGs.* As depicted by the left part of Figure 3.5, mixing interaction and widget works for standard widgets that use a mono-event interaction. For testing *ad hoc* widgets using several multi-event

interactions, EFGs should explicitly describe how user interactions work. For instance, the right part of Figure 3.5 precisely specifies the events that compose the interaction performed by a user (here a multi-touch interaction) to rotate shapes or to cancel the interaction: two touch pressures followed by a move and a touch release to rotate shapes; canceling the interaction in progress consists of pronouncing the word "cancel". In this case, EFGs would be able to support *ad hoc* interactions.

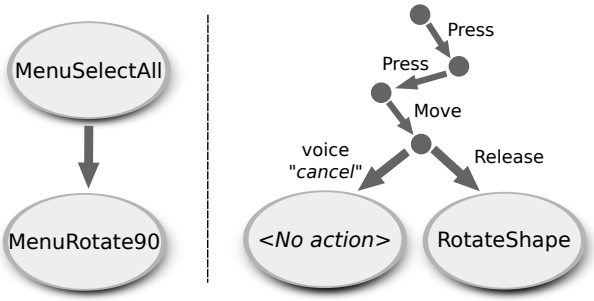


Figure 3.5: EFG sequence on the left. Interaction-action sequence on the right.

We explained in this section the precise limits of the current GUI model-based testing approaches for testing advanced GUIs. We claim that these limitations stem from two facts: first, there is a lack of proper abstractions to build test model for testing advanced GUIs; second, only few GUI oracles are currently considered while we demonstrated in Section 3.1 the diversity of faults that can affect GUIs.

## 3.7 GUI testing with richer UIDLs

This section details how we leveraged a rich User Interface Description Language (UIDL), called Malai [Blo09, BB10, BMN+11], to test advanced GUIs. Our goal is to propose the concept of interaction-action-flow graph (IFG) for going beyond the current limits of EFGs (Section 3.6). First, we introduce the Malai UIDL. Then, the concept of the IFG is explained. Finally, we present the initial feedback from the use of the proposed IFG in two use cases.

### 3.7.1 Modeling GUIs with Malai UIDL

This section illustrates the expressiveness required to build test models of advanced GUIs using Malai. We notably show how to model both standard and advanced interactions, and how to differentiate an interaction from the resulting action of its execution. Malai is an architectural design pattern that extends MVC/MVP by refining the controller to consider the notion of interactions, actions, and instruments as first-class objects [Blo09, BB10]. Malai gathers principles of several major interaction models

and design patterns, notably the *instrumental interaction* [BL00], the *direct manipulation* [Shn83], and the *Command* and *Memento* design patterns [GHJV95]. The Malai implementation[11] provides a model-based UIDL.

Malai decomposes an interactive system as a set of presentations and instruments (see Figure 3.6). A presentation is composed of an abstract presentation and a concrete presentation. An abstract presentation is the data model of the system (the *model* in MVC). A concrete presentation is a graphical representation of an abstract presentation (the *view* in MVC).

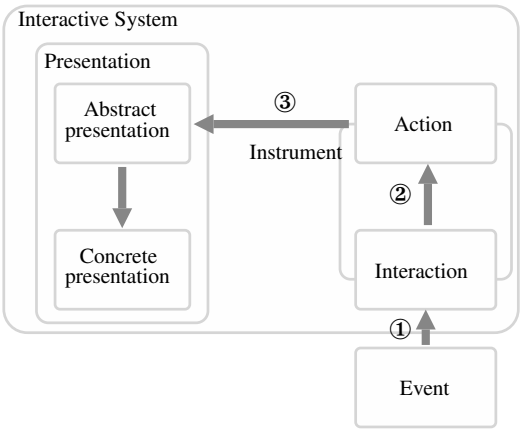


Figure 3.6: The architectural design pattern Malai

An action encapsulates what users can modify in the system. For instance, LaTeXDraw (recall Section 3.4.1), has numerous actions such as *rotating shapes*. An action does not specify how users have to interact with the system to perform it. An action just specifies the results of a user interaction on the system. An action can also depend on other actions for being executed. For example, the action *paste* can be executed only if an action *copy* or *cut* has been executed before. An action is executed on a presentation (link ③).

An interaction is represented by a finite-state machine where each transition corresponds to an event produced by an input device (Figure 3.6, link ①). Using FSMs for defining interactions permits the conception of structured multi-event interactions, such as DnD, multi-touch, or multi-modal interactions. An interaction is independent of any interactive system that may use it. For instance, a bi-manual interaction, as depicted by Figure 3.7, is defined as an FSM using events produced by pointing devices or speech recognizers (*e.g.*, pressure, voice). This interaction does not specify the actions to perform on a system when executed (*e.g.*, rotate shapes). The interaction depicted by Figure 3.7 starts at the initial state (black circle) and ends when entering a terminal (double-lined circle) or an aborting (crossed out circle) state. Aborting states permit users to abort the interaction they are performing. Transitions (*e.g.*, *press* and *move*) can be supplemented with a condition constraining the trigger of the transition. For

[11] `https://github.com/arnobl/Malai`

instance, the interaction goes into the aborting state thanks to the transition *voice* only if the pronounced word is "*abort*".

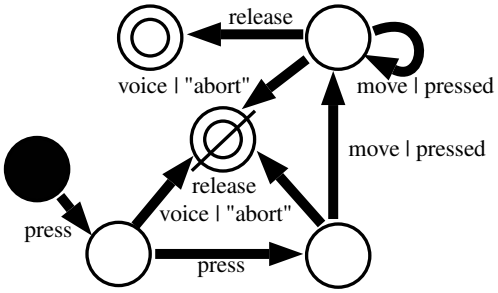


Figure 3.7: Example of a bi-manual interaction modelled by Malai UIDL

Because actions and interactions are independently defined, the role of instruments is to transform input interactions into output actions (link ②). Instruments reify the concept of tool one uses in her every day life to manipulate objects [BL00]. For instance, Figure 3.8 describes the instrument *Hand* as it could have been defined in Latexdraw. The goal of this instrument is to move and rotate shapes. Performing these actions requires different interactions: rotating shapes can be done using the bimanual interaction previously depicted; moving shapes can be done using a DnD interaction. In Malai such tuples interaction-action are called *interactors* and one instrument can have several interactors, *i.e.*, one can handle an instrument with different interactions to execute different actions. In an interactor, the execution of the action is constrained by a condition. For example, the interactor *Bimanual2Rotate* (Figure 3.8) permits the execution of the action *RotateShapes* only if the source and target objects of the bi-manual interaction are the same shape: $src == tgt \wedge src\ is\ Shape$ (using a bi-manual interaction to rotate a shape may imply that the two pressures are done on the targeted shape).

In Malai, the GUIs of a system are composed of the widgets provided by all the instruments: an interaction can be based on a widget (*e.g.*, a click on a button) and in this case, instruments using such interactions create and provide these widgets.

Malai supports mono-event as well as multi-event interactions; interactions work as FSM that may help the test model generation process; actions can be defined and dependencies between them can be specified; instruments and their interactors can be viewed as FSMs as well. Because in Malai interactions, interactors, and instruments are FSMs, the creation of test models is eased as detailed in the next subsection. Such FSM models capture all the possible scenarios that users can do while interacting with the system. A path in such FSMs corresponds to one scenario.

### 3.7.2 Interaction-action-flow graph

This section introduces the concept of interaction-action-flow graph (IFG) that can be automatically inferred from Malai models. As depicted in Figure 3.9, an IFG follows

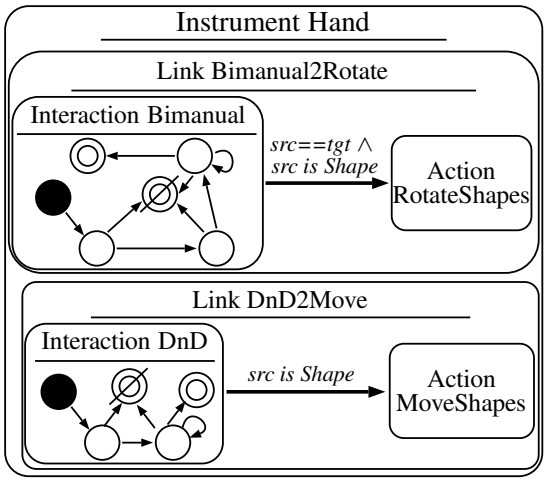


Figure 3.8: Illustration of a Malai instrument

the same idea than an EFG by sequencing all the possible user interactions. The difference is that the concepts of interaction, action, and widget are clearly separated, and interactions and actions are included in IFGs. The goal of such separation is to be able to test a widget using its different interaction, and to test that the effective result of one execution of an interaction is the result expected ass defined in the action. So, an IFG is a sequence of interaction-to-action tuples. Each tuple is in fact an FSM composed of two states (the interaction and the action) and one transition (the condition that permit to execute the action). For instance, Figure 3.9 describes an IFG composed of the three instruments: the *Pencil* (①) that permits to draw shapes; the *EditingSelector* (②) that permits to select the kind of shapes to draw; the *Hand* (③) that permits to move or scale shapes. At start-up, only the instruments *Pencil* and *EditingSelector* are activated and can be used. So, the interaction-action tuples ① and ② are available. The instrument *EditingSelector* permits to use either the *Pencil* or the *Hand* instrument. So, using this instrument can lead to all the tuples ①, ②, and ③. Figure 3.9 does not represent the widgets associated to each tuple but the tuples ① and ③ deal with the drawing area while the tuple ② deals with a set of buttons.

To create such sequences from Malai models, we need to define which instruments can be used at a given instant. Malai models do not explicitly specify the relations between instruments, *i.e.*, they do not specify which instruments can be handled after having used an instrument. Such relations are mandatory to obtain from Malai models a graph of instruments from which test scripts can be produced.

Inferring these relations is automatically performed by analyzing the Malai actions: there exists actions that activate or inactivate instruments. For instance with Figure 3.9, the instrument *EditingSelector* is dedicated to select (*i.e.*, to activate/inactivate) between the instruments *Hand* or *Pencil*. Thus, after having used the *EditingSelector* instrument, either the instruments *Hand* or *Pencil* can be used (transitions ①). Similarly,

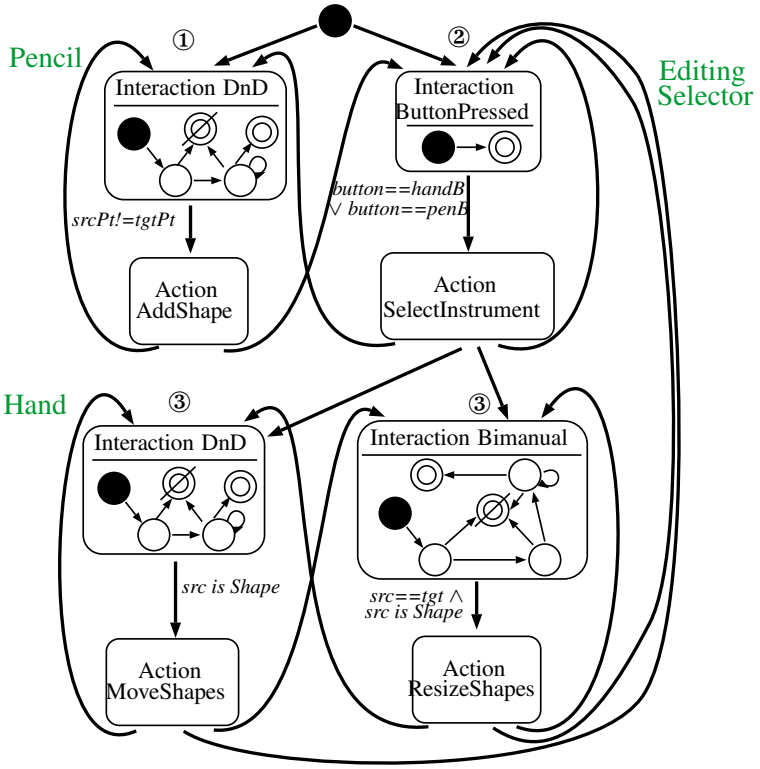


Figure 3.9: Example of an interaction-action-flow graph

after having used the instrument *Hand*, respectively *Pencil*, the instrument *EditingSelector* can be used only (*Pencil*, respectively *Hand*, is inactivated, transitions ②).

Based on the IFG, the test cases can be generated to test both standard and advanced GUIs. For instance, we were able to test multiple *ad hoc* interactions used on advanced widgets. Also, actions permitted to compare the effective result of interacting with SUTs and the expected one. We do not provide, however, a fully automated framework as the case of GUITAR. Instead, we manually built Malai models and manually executed the test scripts.

In the next subsection, we present the benefits of this approach through two different use cases and describe research challenges to overcome.

### 3.7.3 Case studies

To illustrate the expected benefits of the use of Malai and the concept of IFG, we applied our method on two different use cases:

1. **LaTeXDraw** (3.0-alpha1): introduced in Section 3.4.1;
2. **the CONNEXION project**: an industrial project that aims at improving model-based GUI testing techniques.

### LaTeXDraw

Table 3.9 gives an overview of the defects found during our experiment on LaTeXDraw. The manual execution of the tests led to the detection of four defects. All of them were not reported in the official bug trackers of LaTeXDraw. These defects concern different parts of the system.

Table 3.9: Defects found by applying the proposed approach on LaTeXDraw

| Id | Link | Origin |
|---|---|---|
| #1 | `https://bugs.launchpad.net/latexdraw/+bug/788712` | Interaction |
| #2 | `https://bugs.launchpad.net/latexdraw/+bug/768344` | Interaction |
| #3 | `https://bugs.launchpad.net/latexdraw/+bug/770726` | data model |
| #4 | `https://bugs.launchpad.net/latexdraw/+bug/770723` | Java Swing |

Defects #1 and #2 have detected that several user interactions did not work as expected. These user interactions are a multi-click and a DnD interactions. Both of them are *ad hoc* interactions developed specifically for LaTeXDraw. With a multi-click interaction, users can click is several locations in the drawing area to create a shape composed of several points. In our case, executing the interaction did not perform the excepted action, *i.e.*, the creation of the shape. The DnD interaction can be aborted. It means that while executing the DnD, the user can press the key "*escape*" to stop the interaction and to not execute the associated action. These two defects cannot be find by GUITAR since: they have been provoked by non-standard interactions; the two interactions can be executed on the drawing area and we explained that GUITAR cannot support multiple interactions for a same widget.

Defect #3 was an issue in the data model where changes in the Latexdraw's preferences were not considered.

Defect #4 was not a LaTeXDraw issue but a Java Swing one: clicking on the zoom buttons had not effect if performed too quickly. Still, this defect has been detected thanks to the action that zooms on and out the drawing area.

### Industry: feedback from the CONNEXION project

The CONNEXION[12] project is a national industry-driven work program to prepare the design and implementation of the next generation of instrumentation and control (I&C) systems for nuclear power plants. One goal of this project is the development of model-based techniques for testing GUIs of such I&C systems, more precisely for:

- automating as far as possible the current error-prone, expensive, and manual GUI testing process;
- finding GUI errors as early as possible in the development phases to reduce the development cost;

[12] `http://www.cluster-connexion.fr/`

- finding various kinds of GUI errors (not only crashes) on various kinds of *ad hoc* widgets.

The main constraint, imposed by norms [IEC95, IEC10], is that the testing process of I&C systems must be driven from their specifications. So, producing entire GUI models using reverse engineering techniques from the SUT is not possible (some specific information, however, can be extracted from GUIs to be used in GUI models as discussed below).

Similarly to the previous case study (*i.e.*, LaTeXDraw), we manually design GUI models from the provided specifications using Malai. We then automatically generate abstract GUI tests. These last are then concretized to produce executable GUI tests that run on top of the *SCADE LifeCycle Generic Qualified Testing Environment* (SCADE QTE), a tool for testing GUIs developed with the SCADE Display technologies[13]. The concretization phase includes two steps:

- Mapping abstract test cases to the targeted testing framework and SUT. This step requires manual operations to map GUI model elements to their correspond elements in the GUI under test. For instance, the name of the widgets of the GUI under test have to be mapped to the widgets defined in the GUI models. To avoid this step, the specifications have to precisely specify the names to use, as discussed later in this section.
- Generating GUI oracles. We currently generate basic oracles that checks the correct GUI workflow when interacting with widgets.

These preliminary results highlighted the capability of the IFG, based on Malai UIDL, produces test cases capable of detecting some defects in advanced GUIs. However, if the Malai expressiveness permits to test advanced GUIs, we face two UI testing challenges relative to the concretization phase of MBT process (see Figure 2.6 in Chapter 2):

**GUI oracles generation.** GUI oracles have to be produced as far as possible from GUI models. Our proposed fault model describes several faults that imply different ways to detect them. GUI faults imply the development of multiple GUI oracles based on techniques diametrically different. For instance, the graphical nature of GUIs requires their graphical rendering to be checked. To do so, a possible oracle consists of comparing screenshots of GUIs to detect differences. This oracle thus uses image processing techniques. However, checking that a widget is correctly activated can be done using classical code unit testing techniques. These differences between GUI oracles complexifies the GUI model-based testing process that has to generate such oracles. One challenge to tackle is *how to write and generate GUI oracles to detect such failures?* New GUI testing techniques have thus to be developed to automate the generation of GUI oracles being able to detect those failures other than standard GUI failures or crashes.

[13] `http://www.esterel-technologies.com/`

**Incomplete, ambiguous GUI specifications.** The specifications we use to build GUI models do not formally specify in detail the GUI to test. As a result, the GUIs and the GUI models, produced independently each other from the specifications, may differ. For instance, the position of widgets are not explicitly precised. Testers and developers have to determine these positions from GUI mockups. To limit this problem in the CONNEXION project, some information are extracted from the developed GUIs to be reused in the GUI testing models. These extracted information are notably the position of the widgets. It means that the position of the widgets is not tested but is used to detect other GUI errors (*e.g.*, widget responsiveness). This problem of incomplete, ambiguous GUI specifications has been already reported in previous work on model-based testing. Dalal *et al.*, for instance, reported that "*some of the constraints were not covered in the requirements and we had to run the application to see exactly how it had been implemented*" [DJK+99].

## 3.8 Conclusion

This chapter proposes a GUI fault model for providing GUI testers with benchmark tools to evaluate the ability of GUI testing tools to detect GUI failures. This fault model has been empirically assessed by analyzing and classifying into it 279 GUI bug reports of different open-source GUIs. We have used our fault model to evaluate the limits of GUI testing frameworks. The experiments have shown that if current GUI testing tools have demonstrated their ability for finding several kinds of GUI failures, they also fail at detecting several GUI faults we identified. The underlying reasons are twofold. First, GUI failures may be related to the graphical rendering of GUIs. Testing a GUI rendering is a complex task since current testing techniques mainly rely on code analysis that can hardly capture graphical properties. Second, the current trend in GUI design is the shift from designing GUIs composed of standard widgets to designing GUIs relying on more complex interactions and *ad hoc* widgets. Instead, current GUI testing frameworks rely on the concepts of standard widgets and their simple interactions to provide an automated GUI testing.

New GUI testing techniques have thus to be proposed for fully GUI testing, as automated as possible, GUI rendering and complex interactions using *ad hoc* widgets. For this purpose, our fault model can assist GUI testing techniques to cover more GUI faults. We have provided 65 GUI mutants planted into standard and *ad hoc* widgets of a highly interactive GUI system to forge several GUI faults. These mutants are freely available and thus can be used by testing community to achieve that purpose. For example, the third experiment, we run on GUITAR and Jubula, has shown that 43 out of 65 GUI mutants remained alive. Most of them concern the graphical data rendering of *ad hoc* widgets and their complex interactions. These results help their developers to understand which kinds of GUI faults were not detected by their GUI testing tools.

# Chapter 4

# GUI Design Smells: The Case of *Blob listener*

In Chapter 3, we presented a new GUI fault model that describes faults specific for GUIs. Numerous GUI testing techniques have been proposed to detect errors that affect GUI systems. Besides, GUI errors may be accentuated by the low quality of GUI code. For example, the presence of design smells has been correlated to the introduction of faults [LS07, HZBS14] or other aspects of code quality such as maintainability [SYA+13]. Static analyses are developed to find defects in code or to measure the code quality. To contribute to the code assurance quality of GUIs we focus on studying GUI design smells.

This chapter presents a novel static analysis technique to automatically detect a new type of GUI design smell, the *Blob listener*. In particular, we have designed new heuristics that enable to search and locate *Blob listeners* in the source code. The main challenge of this analysis is to infer the code regions related to GUI and GUI controllers and to automatically locate the GUI commands handled by these controllers. The most critical step is to design and implement a heuristic that accurately detects GUI commands in GUI listeners. The reasons are twofold. First, GUI listeners may have conditional blocks that do not handle events produced by widgets, *i.e.*, non-GUI commands. Also, we have identified three variants of *Blob listeners* that are different ways to identify the widget that produced the event: 1. comparing a property of the widget; 2. checking the type of the widget; and 3. comparing widget references. Second, GUI commands can be nested within other commands. To overcome this step, we have implemented an algorithm that infers recursively all references in conditional statements and determines whether they refer to a GUI object. To manage the case of the nested commands, our algorithm removes the irrelevant ones based on that variants.

We implement our static analysis in a tool called `InspectorGuidget`. This tool is an Eclipse plug-in publicly available[1] dedicated to Java software systems based on the Swing toolkit. The ability of our tool to detect *Blob listeners* is evaluated on six representative highly interactive software systems. These systems cover several user

[1] `https://github.com/diverse-project/InspectorGuidget`

interactions implemented in different kind of GUI listeners. To build a ground truth for our experiments we manually retrieved all instances of *Blob listeners* in each application. We then run our tool against each application to detect GUI listeners, commands, and *Blob listeners*.

In this chapter, we first introduce the motivation of studying GUI controllers and explain why we have identified a *Blob listener* as a GUI design smell (Section 4.1). We then present the empirical study to investigate the current development practices regarding listeners in GUI controllers (Section 4.2). Based on this study, we identify and characterize a *Blob listener* as a GUI design smell presented in Section 4.3. We then propose a systematic static code analysis procedure to search for *Blob listeners* (Section 4.4). Next, we evaluate our approach and present the results (Section 4.5). The threats to validity of the experiments are presented in Section 4.6. We also discuss the scope of `InspectorGuidget` and the good coding practices we proposed to avoid the use of *Blob listeners* (Section 4.7). This chapter ends with the preliminary steps towards refactoring *Blob listeners* (Section 4.8).

## 4.1 Introduction

Software engineers develop GUIs following widespread architectural design patterns that consider GUIs as first-class concerns (recall Chapter 2). GUI implementations rely on event-driven programming where events are handled by controllers (or presenters[2]). Listing 4.4 illustrates a simple example of GUI controller in Java Swing code. In this example, the *AController* manages events produced by two widgets, *b1* and *b2* (Lines 2–3). To handle events these widgets trigger in response of users' interactions, *AController* implements the *ActionListener*. The code that reacts to the events is included in the GUI listener *actionPerformed*. A GUI command, *i.e.*, a set of statements executed in reaction of a GUI event, is produced for each widget (Lines 8 and 10). We define a *Blob listener* as a GUI listener that can produce multiple GUI commands.

```
1  class AController implements ActionListener {
2     JButton b1;
3     JButton b2;
4
5     @Override public void actionPerformed(ActionEvent e) {
6        Object src = e.getSource();
7        if(src==b1){
8           // Command 1
9        }else if(src==b2)
10          // Command 2
11       }
12    }
13    //...
14 }
```

Listing 4.1: Example of a GUI controller in Java Swing code

[2]For simplicity, we will use the term controller throughout this chapter to refer to any kind of component of MV* architectures that manage events triggered by GUIs, such as Presenter (MVP), or ViewModel (MVVM [Smi09]).

However, the use of *Blob listeners* in real-word GUI implementations complexifies the GUI code for the following reasons.

1. *Use of multiple conditional statements.* *Blob listeners* handle multiple GUI commands. Commands use conditional statements (*e.g.*, *if-then-else*) to identify the widget that produced an event. Also, the conditional statements may be used or added to check a particular class type of a widget (*e.g.*, JButton or JCheckbox). So, if listeners are implemented as *Blob listeners*, this implementation also collaborates to increase the cyclomatic complexity of a GUI code. A code with high CC is more error-prone [KMHSB10] and harder to maintain [BM11, GZDG15]. An example of a GUI listener code excerpt from the Github repository is shown in Listing 5.4 (see Appendix). This listener has a high cyclomatic complexity (CC value is 174) and manages several widgets. We have identified 39 GUI commands in this listener, which characterizes a *Blob listener*. One may note that a high cyclomatic complexity of a listener indicates that it is a candidate to be a *Blob listener*. Another point is that if developers use a single listener to manage several widgets (which is the case of Listing 5.4), this listener may grow significantly and thus more difficult to be read (*e.g.*, multiple *if-then-else* statements that can be very large) and maintained (*e.g.*, time-consuming changes if the GUI evolves often).

2. *Similar code within a GUI listener.* *Blob listeners* may contain the same or similar code structure in more than one GUI command. We have observed this practice in GUI listeners that implement commands concerning opposite actions such as *undo/redo* or *zoom in/out* actions. Listing 4.2 illustrates this case. The two *if* blocks contain a set of statements to be executed in reaction of "*zoom in*" (Lines 4–10) and "*zoom out*" (Lines 12–18) actions. These blocks represent two commands (Command #1 and #2), which share five out of six statements. Methods with duplicated code have been associated with maintainability problems for instance leading to more faults or more change effort [SYA+13].

```
1  @Override public void actionPerformed(ActionEvent e){
2  //...
3      if(label.equals("LABEL_ZOOM_IN")) {//Command #1
4        drawPanel.zoomIn();//non-shared statement
5        drawPanel.getFrameParent().getToolbar().getZoomField().setValue(...);
6        mainFrame.getXScale().repaint();
7        mainFrame.getYScale().repaint();
8        mainFrame.requestFocus();
9        return;
10     }
11     if(label.equals("LABEL_ZOOM_OUT")){//Command #2
12       drawPanel.zoomOut();//non-shared statement
13       drawPanel.getFrameParent().getToolbar().getZoomField().setValue(...);
14       mainFrame.getXScale().repaint();
15       mainFrame.getYScale().repaint();
16       mainFrame.requestFocus();
17       return;
18     } //... more than 50 GUI commands
```

Listing 4.2: Example of a quite similar code within a *Blob listener*

3. *Complex and large conditional expressions.* We have observed different implementations in *Blob listeners* to identify which widgets produced the event. The typical implementations leverage the widget attribute as exemplified by Lines 7 and 9 in Listing 4.1. Other implementations, however, do not handle directly this attribute. Instead, they leverage the properties of widgets (*e.g.*, getActionCommand) or other attributes to access the target widget. These kinds of implementations require a semantic code analysis to infer recursively their complex and large expressions to obtain for instance a reference to a GUI object. The examples of such expressions in *Blob listeners* are as follows:

```
if(label.equals("LABEL_DRAW_X_LABELS") || label.equals("LABEL_DRAW_Y_LABELS"))...

if (!BaseEditDelegate.getEditMode(m_gameData)
  && (!(e.isControlDown() || e.isAltDown() || e.isShiftDown())
  && e.getButton() == MouseEvent.BUTTON1
  && (m_clickedAt != null
  && m_releasedAt != null)))...

if ((evt.getX() >= center.pictureScrollPane.getViewport().getWidth())
  || (evt.getY() >= center.pictureScrollPane.getViewport().getHeight())
  || (evt.getX() <= 0)
  || (evt.getY() <= 0)
  || (evt.getX() > (w*zoom))
  || (evt.getY() > (h*zoom)))...
```

4. *Faults.* Faults have been reported into *Blob listeners*. One example is the two issues reported in a code repository[3] from Github. Listing 4.3 shows the excerpt of the GUI faulty code.

```
1  public class JWhiteBoard extends ReceiverAdapter
2                  implements ActionListener, ChannelListener {
3     //..
4     clearButton=new JButton("Clean");
5     clearButton.addActionListener(this);
6     leaveButton=new JButton("Exit");
7     leaveButton.addActionListener(this);
8     //...more than 150 lines of code
9
10    @Override public void actionPerformed(ActionEvent e) {
11        String command=e.getActionCommand();
12         if("Clear".equals(command)) {//GUI fault in Command #1
13             if(noChannel) {
14                 clearPanel();
15                 return;
16             }
17             sendClearPanelMsg();
18         }
19         else if("Leave".equals(command)) {//GUI fault in Command #2
20             stop();
21         }//...
22     }//...
23 }
```

Listing 4.3: Example of GUI faults within a *Blob listener*

[3] https://github.com/LinhTran95/Asg7--Team10/

The listener method "*actionPerformed*" (Lines 10–22) handles two commands: Command #1 (Lines 12–18) and Command #2 (Lines 19–21). Command #1 and Command #2 have a set of statements to be executed when the widgets labeled "*Clear*" and "*Leave*" are pressed, respectively. However, GUI failures will occur when users interact with such widgets. The two GUI faults are located in the conditional expressions of that commands (Lines 12 and 19) used to check which widgets produced the event. The strings associated to identify each widget are incorrect. The correct strings are "*Clean*" and "*Exit*" (Lines 4 and 6) instead of "*Clear*" and "*Leave*", respectively. Both GUI faults are related to the category *Action* of our fault model (recall Table 3.2 in Chapter 3).

The above scenarios highlight our findings in the implementations of GUI controllers. These findings provide a strong motivation for characterizing *Blob listeners* as a new kind of GUI design smells, and for the development of tools that detect them. Indeed, a *Blob listener* is a GUI specific instance of a more general bad smell: *God method* [BMMM98], that characterizes methods that "*know too much or do too much*".

In the next sections, we present the systematic approach to identify and to detect automatically *Blob listeners* in GUI controllers.

## 4.2 What Compose GUI listeners? An Empirical Study

In this section, we present the empirical study to identify current coding practices in Java implementations of GUI controllers. We also present how this study led us to identify and characterize a *Blob listener* as a new GUI design smell.

As any other code artifact, the code of GUI controllers may be affected by good and bad coding practices. In most widespread GUI toolkits (*e.g.*, in Java Swing, JavaFX[4], GWT[5], and SWT[6]) controllers rely on GUI listeners for treating events triggered by widgets. In this section, we provide empirical observations about how such GUI listeners are coded in legacy Java Swing code[7]. In particular, we focus on the following research question:

**RQ0** Is there any difference in term of code metrics between GUI listener methods and methods that are not listeners?

This preliminary analysis focuses on the Java Swing toolkit because of its popularity and the large quantity of Java Swing legacy code stored on various Web code hosting services (*e.g.*, `github.com` or `sourceforge.net`). In this study, we collected 511 code repositories from *Github* that contain references to Java Swing in their description[8].

[4] `http://docs.oracle.com/javase/8/javase-clienttechnologies.htm`

[5] `http://www.gwtproject.org/`

[6] `https://www.eclipse.org/swt/`

[7] The sources used in our study and the resulting data are available here: `https://github.com/diverse-project/InspectorGuidget`

[8] The following Github query has been used: `https://api.github.com/search/repositories?q=swing+GUI+in:description,readme+language:java&sort=stars&per_page=100`. Because of resource constraints, only the 511 most starred repositories out of the 1200 available have been analyzed.

Java Swing provides a set of GUI listeners that controllers can implement. To identify GUI listener methods, we search for all the methods that match a Java Swing listener method. The 511 repositories contain 1 935 262 lines of Java code (LoC): 16 617 listener methods (3.6 % of the total LoCs) and 319 795 non-listener methods (96.4 % of LoCs). To obtain an overview of the structure and the complexity of each Java method, we collected the cyclomatic complexity (CC) and the number of control flow statements. Empty methods are ignored.

The results are provided in Table 4.1. We performed an independent samples Mann-Whitney test [She07] (data do not follow a normal distribution) to compare the data gathered from listener and non-listener methods, using a 95 % confidence level (*i.e.*, $p$-value$<0.05$). We observe important variations for all the measured metrics, between listener and non-listener methods. *Switch* and *loop* statements per statement are more than 62 % to 68 % less used in listeners than in non-listener methods. Similarly, the mean CC of listeners is 16.4 % less than the mean CC of non-listener methods. Meanwhile, there are 24.3 % more *if* statements per statement in listeners than in non-listener methods. All the $p$-values below 0.05 indicate that the variations we observed are statistically significant and not due to randomness. Based on these observations, we conclude that Java Swing GUI listener methods are not developed as other methods.

Table 4.1: Statistics per listener and non-listener methods

| **Code Metrics** | **Listeners (Mean)** | **Non-listeners (Mean)** | **Diff. (%)** | **Significance $p$-value** |
|---|---|---|---|---|
| If (# per stmt) | 0.074 | 0.056 | 24.32 | $< 0.0001$ |
| Switch (# per stmt) | 0.003 | 0.008 | −62.5 | $< 0.0001$ |
| Loop (# per stmt) | 0.005 | 0.016 | −68.7 | $< 0.0001$ |
| Statements (#) | 4.871 | 6.389 | −23.76 | $< 0.0001$ |
| CC | 2.166 | 2.591 | −16.40 | $< 0.0001$ |

Now, focusing on *if* statements, Figure 4.1 gives the distribution of the number of listeners according to their number of *if* statements. Most of listeners have less than three *if* statements (91.7 %, *i.e.*, 13 227 of the 14 420 non-empty analyzed listeners). We also looked at the kinds of listeners that use more than two *if* statements in their code (8.3 %). 47.3 % (*i.e.*, 564) of these listener methods are concrete implementations of *actionPerformed* declared in the *ActionListener* interface (used to receive events from triggerable widgets such as buttons, check-boxes, and menus).

In the next section we explain why listeners have a higher use of *if* statements than other methods.

## 4.3 *Blob listener*: Definition & Illustration

This section introduces the *Blob listener* and illustrates some bad practices found in the projects we retrieved from Github.

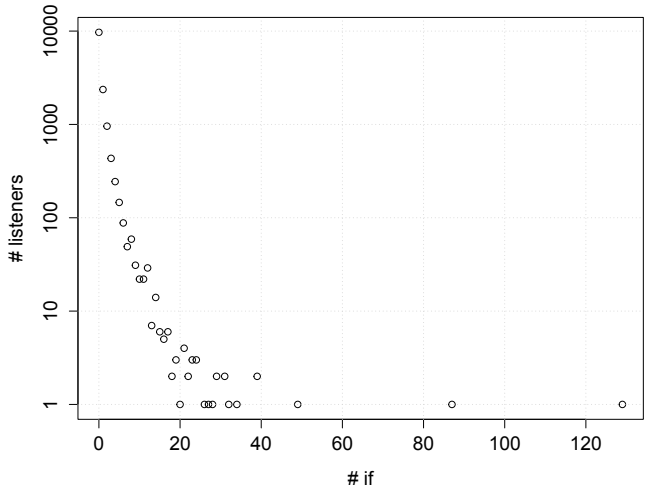


Figure 4.1: Distribution of the listeners according to their number of *if* statements

### 4.3.1 *Blob listener*

The previous analysis shows that some GUI listener methods make an intensive use of conditional statements. We pored over these listeners and identified one recurrent bad practice that developers use while coding GUI listeners using Java Swing. We call this bad practice the *Blob listener*, as defined as follows:

**Definition 4.1 (GUI Command)** *A GUI command [GHJV95, BL00], aka.* action *[BB10], is a set of statements executed in reaction to a user interaction, captured by an input event, performed on a GUI. GUI commands may be supplemented with: a pre-condition checking whether the command fulfils the prerequisites to be executed; undo and redo functions for, respectively, canceling and re-executing the command.*

**Definition 4.2 (*Blob listener*)** *A* Blob listener *is a GUI listener that can produce several GUI commands.* Blob listeners *can produce several commands because of the multiple widgets they have to manage. In such a case, Blob listeners' methods (such as actionPerformed) may be composed of a succession of conditional statements that:*

1. *Check whether the widget that produced the event to treat is the good one, i.e., the widget that responds a user interaction; and*
2. *Execute the command when the widget is identified.*

We identified three variants of the *Blob listener* that are detailed in the next subsection.

### 4.3.2 Variants of *Blob listener*

The variations of a *Blob listener* reside in the way of identifying the widget that produced the event. These three variants are described and illustrated as follows.

**Comparing a property of the widget.** Listing 4.4 is an example of the first variant of *Blob listener*: the widgets that produced the event (lines 10, 13, and 15) are identified with a *String* associated to the widget and returned by *getActionCommand* (line 9). Each of the three *if* blocks defines a behavior depending on the triggered widget (lines 11, 12, 14, and 16). Each block of instructions forms a *command*.

```
1  public class MenuListener
2                implements ActionListener, CaretListener {
3   //...
4   protected boolean selectedText;
5
6   @Override public void actionPerformed(ActionEvent e) {
7    Object src = e.getSource();
8    if(src instanceof JMenuItem || src instanceof JButton){
9       String cmd = e.getActionCommand();
10      if(cmd.equals("Copy")){//Command #1
11        if(selectedText)
12          output.copy();
13      }else if(cmd.equals("Cut")){//Command #2
14         output.cut();
15      }else if(cmd.equals("Paste")){//Command #3
16        output.paste();
17      }
18      // etc.
19     }
20   }
21
22  @Override public void caretUpdate(CaretEvent e){
23     selectedText = e.getDot() != e.getMark();
24     updateStateOfMenus(selectedText);
25  }
26 }
```

Listing 4.4: Widget identification using widget's properties in Swing

In Java Swing, the properties used to identify widgets are mainly the *name* or the *action command* of these widgets. The action command is a string used to identify the kind of commands the widget will trigger. Listing 4.5, related to Listing 4.4, shows how an action command (lines 2 and 6) and a listener (lines 3 and 7) can be associated to a widget in Java Swing during the creation of the user interface.

```
1 menuItem = new JMenuItem();
2 menuItem.setActionCommand("Copy");
3 menuItem.addActionListener(listener);
4
5 button = new JButton();
6 button.setActionCommand("Cut");
7 button.addActionListener(listener);
8 //...
```

Listing 4.5: Initialization of Swing widgets to be controlled by the same listener

**Checking the type of the widget.** The second variant of *Blob listener* consists of checking the *type* of the widget that produced the event. Listing 4.6 depicts such a

practice where the type of the widget is tested using the operator *instanceof* (Lines 3, 5, 7, and 9). One may note that such *if* statements may have nested *if* statements to test properties of the widget as explained in the previous point.

```
1  public void actionPerformed(ActionEvent evt) {
2     Object target = evt.getSource();
3     if (target instanceof JButton) {
4        //...
5     } else if (target instanceof JTextField) {
6        //...
7     } else if (target instanceof JCheckBox) {
8        //...
9     } else if (target instanceof JComboBox) {
10       //...
11    }
12 }
```

Listing 4.6: Widget identification using the operator *instanceof*

**Comparing widget references.** The last variant of *Blob listener* consists of comparing widget references to identify those at the origin of the event. Listing 4.7 illustrates this variant where *getSource* returns the source widget of the event that is compared to widget references contained by the listener (*e.g.*, lines 2, 4, and 6).

```
1  public void actionPerformed(ActionEvent event) {
2     if(event.getSource() == view.moveDown) {
3        //...
4     } else if(event.getSource() == view.moveLeft) {
5        //...
6     } else if(event.getSource() == view.moveRight) {
7        //...
8     } else if(event.getSource() == view.moveUp) {
9        //...
10    } else if(event.getSource() == view.zoomIn) {
11       //...
12    } else if(event.getSource() == view.zoomOut) {
13       //...
14    }
15 }
```

Listing 4.7: Comparing widget references

In these three variants, multiple *if* statements are successively defined. Such successions are required when one single GUI listener gathers events produced by several widgets. In this case, the listener needs to identify the widget that produced the event to process.

The three variants design smell also appear in others Java GUI toolkits, namely SWT, GWT, and JavaFX. Examples for these toolkits are available on the webpage of this work[1].

## 4.4 Detecting *Blob listeners*

In this section, we present our static analysis approach to automatically detect *Blob listeners*.

### 4.4.1 Approach Overview

We characterize *Blob listeners* as GUI listeners that can produce several GUI commands. Section 4.2 showed empirical evidence that listeners mainly use conditional statements to identify widgets. Figure 4.2 describes the process we propose to automatically detect *Blob listeners*. The detection process includes three main steps:

1. *GUI listeners detection*: GUI listeners that contain conditional blocks (conditional GUI listeners) are automatically detected in the source code through a static analysis (Section 4.4.2);

2. *GUI commands detection*: the GUI commands, produced while interacting with widgets, that compose conditional GUI listeners are automatically detected using a second static analysis (Section 4.4.3); and

3. *Blob listeners detection*: the analysis of these commands allows us to spot the GUI listeners that are *Blob listeners* (Section 4.4.4), *i.e.*, those having more than one command.

`InspectorGuidget` uses *Spoon*, a library for transforming and analyzing Java source code [PMP+06], to support the static analyses.

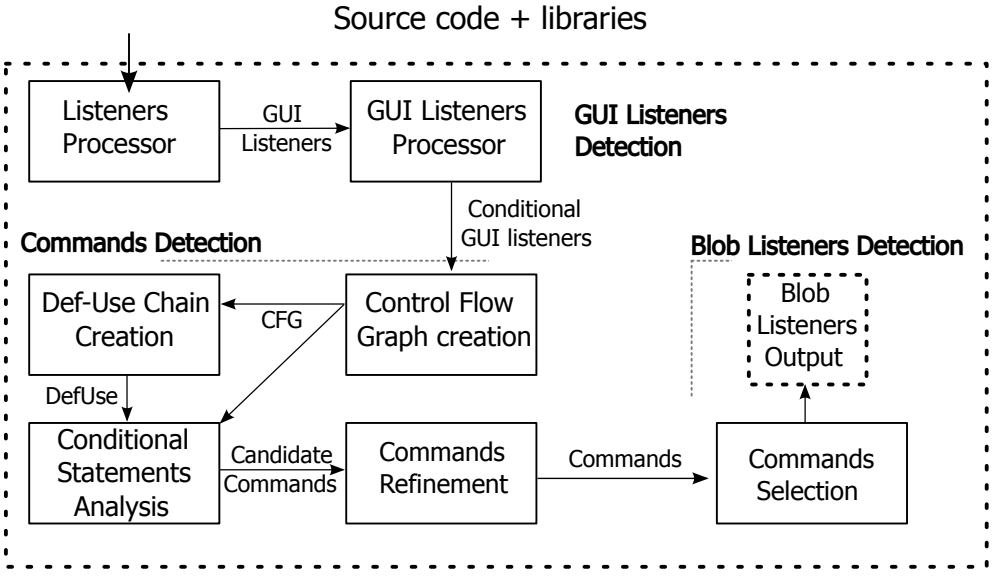


Figure 4.2: The proposed process for automatically detecting *Blob listeners*

### 4.4.2 Detecting Conditional GUI Listeners

We define a conditional GUI listener as follows:

**Definition 4.3 (Conditional GUI listener)** *A conditional GUI listener is a listener composed of conditional blocks used to identify the widget that produced an event to process. Such conditional blocks may encapsulate a command to execute in reaction to the event.*

For example, five nested conditional blocks (Lines 8, 10, 11, 13, and 15) compose the listener method *actionPerformed* in Listing 4.4 (Section 4.3). The first conditional block checks the type of the widget that produced the event (Line 8). This block contains three other conditional blocks that identify the widget using its action command (Lines 10, 13, and 15). Each of these three blocks encapsulates one command to execute in reaction of the event.

Algorithm 1 details the detection of conditional GUI listeners. The inputs are all the classes of an application and the list of classes of a GUI toolkit. First, the source code classes are processed to identify the GUI controllers. When a class implements a GUI listener (Line 5), all the implemented listener methods are retrieved (Line 6). For example, a class that implements the *MouseMotionListener* interface must implement the listener methods *mouseDragged* and *mouseMoved*. Next, each GUI listener is analyzed to identify those having at least one conditional statement (Lines 8 and 9). All listeners with those statements are considered as conditional GUI listeners (Line 10).

**Algorithm 1** Conditional GUI Listeners Detection

**Input:** *classes*, the source classes of the software system
**Input:** *tkClasses*, the classes of the GUI toolkit
**Output:** *listeners*, the detected conditional GUI listeners
1: $GUIListeners \leftarrow \varnothing$
2: $listeners \leftarrow \varnothing$
3: **foreach** $c \in classes$ **do**
4: **foreach** $tkc \in tkClasses$ **do**
5: **if** $c$.isSubtypeOf($tkc$) **then**
6: $GUIListeners \leftarrow GUIListeners \cup getMethods(c, tkc)$
7: **foreach** $listener \in GUIListeners$ **do**
8: $statements \leftarrow getStatements(listener)$
9: **if** hasConditional($statements$) **then**
10: $listeners \leftarrow listeners \cup \{listener\}$

### 4.4.3 Detecting Commands in Conditional GUI Listeners

Algorithm 2 details the detection of GUI commands. The input is a set of GUI conditional listeners. The statements of conditional GUI listeners are processed to detect commands. First, we build the control-flow graph (CFG) of each listener (Line 6). Second, we traverse the CFG to gather all the conditional statements that compose a given

statement (Line 7). Next, these conditionals are analyzed to detect any reference to a GUI event or widget (Line 8). Typical references we found are, for instance:

```
if(e.getSource() instanceof Component)...
if(e.getSource() == copy)...
if(e.getActionCommand().contains("copy"))...
```

where *e* refers to a GUI event, *Component* to a Swing class, and *copy* to a Swing widget. The algorithm recursively analyzes the variables and class attributes used in the conditional statements until a reference to a GUI object is found in the controller class. For instance, the variable *actionCmd* in the following code excerpt is also considered by the algorithm.

```
String actionCmd = e.getSource().getActionCommand()
if("copy".equals(actionCmd)) ...
```

**Algorithm 2** Commands Detection

**Input:** *listeners*, the detected conditional GUI listeners
**Output:** *commands*, the commands detected in *listeners*

```
1:  commands ← getProperCmds(getPotentialCmds(listeners))
2:
3:  function getPotentialCmds(listeners)
4:      cmds = ∅
5:      foreach listener ∈ listeners do
6:          cfg ← getControlFlowGraph(listener)
7:          foreach stmts ∈ cfg do
8:              conds ← getCondStatementsUseEventOrWidget(stmts)
9:              cmds = cmds ∪ {Command(stmts, conds, listener)}
10:     return cmds
11:
12: function getProperCmds(candidates)
13:     nestedCmds ← ∅
14:     notCandidates ← ∅
15:     foreach cmd ∈ candidates do
16:         nestedCmds ← nestedCmds ∪ (cmd, getNestCmds(cmd))
17:     foreach (cmd, nested) ∈ nestedCmds, |nested| > 0 do
18:         if |nested| == 1 then
19:             notCandidates ← notCandidates ∪ nested
20:         else
21:             notCandidates ← notCandidates ∪ {cmd}
22:     return candidates ∩ notCandidates
```

When a reference to a GUI object is found in a conditional statement, it is considered as a potential command (Line 9). These potential commands are then more precisely analyzed to remove irrelevant ones (Lines 12–22) as discussed below.

**Selecting the most representative commands.** A conditional block statement can be surrounded by other conditional blocks. Potential commands detected in the function *getPotentialCmds* can thus be nested within other commands. We define such commands as *nested commands*. In such a case, the algorithm analyzes the nested conditional blocks to detect the most representative command. We observed two cases:

1. *A potential command contains only a single potential command, recursively.* The following code excerpt depicts this case. Two potential commands compose this code. Command #1 (Lines 1–5) has a set of statements (*i.e.*, command #2) to be executed when the widget labeled *"Copy"* is *pressed*. However, command #2 (Lines 2–4) only checks whether there is a text typed into the widget *"output"* to allow then command #1 be executed. So, command #2 works as a precondition to command #1, which is the command executed in reaction to that interaction. In this case, only the first one will be considered as a GUI command.

```
1 if(cmd.equals("Copy")){ //Potential command #1
2   if(!output.getText().isEmpty()) {//Potential command #2
3     output.copy();
4   }
5 }
```

2. *A potential command contains more than one potential command.* The following code excerpt depicts this case. Four potential commands compose this code (Lines 1, 3, 5, and 7). In this case, the potential commands that contain multiple commands are not considered. In our example, the first potential command (Line 1) is ignored. One may note that this command checks the type of the widget, which is a variant of *Blob listener* (see Section 4.3.1). The three nested commands, however, are the real commands triggered on user interactions.

```
1 if(src instanceof JMenuItem){ //Potential command #1
2   String cmd = e.getActionCommand();
3   if(cmd.equals("Copy")){ //Potential command #2
4   }
5   else if(cmd.equals("Cut")){ //Potential command #3
6   }
7   else if(cmd.equals("Paste")){ //Potential command #4
8   }
9 }
```

These two cases are described in Algorithm 2 (Lines 17–21). Given a potential command, all its nested potential commands are gathered (Lines 15–16). The function *getNestCmds* analyses the commands by comparing their code line positions, statements, *etc.* So, if one command $C$ contains other commands, they are marked as nested to $C$. Then, for each potential command and its nested ones: if the number of nested commands equals 1, the single nested command is ignored (Lines 18–19); if the number of nested commands is greater than 1, the root command is ignored (Line 21).

### 4.4.4 *Blob listener* Detection

*Blob listeners* represent all GUI listeners that can produce more than one command. Thus, detecting commands is the crucial step to identify *Blob listeners*. For this reason our command detection algorithm requires to select the most representative commands per listener.

Algorithm 3 describes the *Blob listeners* detection. To detect *Blob listeners*, our algorithm first gathers all the detected commands per GUI listeners (Line 2). These commands are representative since Algorithm 2 removes the irrelevant commands as detailed in Section 4.4.3. Then, if a GUI listener has more than one command, it is marked as a *Blob listener* (Lines 4–5).

**Algorithm 3** *Blob listeners* Detection

**Input:** *commands*, the commands detected by Algorithm 2
**Output:** *blobListeners*, the detected *Blob listeners*
1: $blobListeners \leftarrow \varnothing$
2: $candidatesBlob \leftarrow gatherCommandsByListener(commands)$
3: **foreach** $candidate \in candidatesBlob$ **do**
4: **if** hasCommands($candidate$) **then**
5: $blobListeners \leftarrow blobListeners \cup \{candidate\}$

## 4.5 Evaluation

To evaluate the efficiency of our approach, we address the three following research questions:

**RQ1** To what extent is the detection algorithm able to detect GUI commands in GUI listeners correctly?

**RQ2** To what extent is the detection algorithm able to detect *Blob listeners* correctly?

**RQ3** Are all *Blob listeners* look alike?

The evaluation has been conducted using `InspectorGuidget`, our implementation of the *Blob listener* detection algorithm. `InspectorGuidget` is an Eclipse plug-in that can analyze Java Swing software systems. `InspectorGuidget` leverages the Eclipse development environment to raise warnings in the Eclipse Java editor on detected *Blob listeners* and their GUI commands. Initial tests have been conducted on software systems not reused in this evaluation.

`InspectorGuidget` and all the material of the evaluation are freely available at the following address: `https://github.com/diverse-project/InspectorGuidget`.

### 4.5.1 Objects

We conducted our evaluation by selecting six well-known or large open-source software systems based on the Java Swing toolkit: *FastPhotoTagger*, *GanttProject*, *JAxoDraw*,

*Jmol*, *TerpPaint*, and *TripleA*. For each system, we downloaded the source code and configured its Java project in Eclipse.

Table 4.2 lists the subject systems and Table 4.3 presents their characteristics such as their number of GUI listeners and LOC size.

Table 4.2: Subject interactive systems

| **Software Systems** | **Version** | **Source Repository Link** |
|---|---|---|
| FastPhotoTagger | 2.3 | `http://sourceforge.net/projects/fastphototagger/` |
| GanttProject | 2.0.10 | `https://code.google.com/p/ganttproject/` |
| JaxoDraw | 2.1 | `http://jaxodraw.svn.sourceforge.net/svnroot/jaxodraw/trunk/jaxodraw` |
| Jmol | 14.1.13 | `http://svn.code.sf.net/p/jmol/code/trunk/Jmol` |
| TerpPaint | 3.4 | `http://sourceforge.net/projects/terppaint/files/terppaint/3.4/` |
| TripleA | 1.8.0.3 | `https://svn.code.sf.net/p/triplea/code/trunk/triplea/` |

Table 4.3: Characteristics of the subject interactive systems

| **Software Systems** | **GUI Listeners (# (LOC))** | **Conditional GUI (# (LOC))** |
|---|---|---|
| FastPhotoTagger | 96 (567) | 24 (417) |
| GanttProject | 68 (435) | 14 (282) |
| JaxoDraw | 129 (1347) | 52 (1137) |
| Jmol | 252 (1673) | 53 (1204) |
| TerpPaint | 272 (1089) | 4 (548) |
| TripleA | 580 (6188) | 174 (4321) |

### 4.5.2 Methodology

The accuracy of the static analyses that compose the detection algorithm is measured by the *recall* and *precision* metrics [OPM15]. We ran `InspectorGuidget` on each software system to detect GUI listeners, commands, and *Blob listeners*. We assume as a precondition that only GUI listeners are correctly identified by our tool. Thus, to measure the precision and recall of our automated approach, we manually analyzed all the GUI listeners detected by `InspectorGuidget` to:

- *Check conditional GUI Listeners.* For each GUI listener, we manually checked whether it contains at least one conditional GUI statement. The goal is to answer **RQ1** and **RQ2** more precisely, by verifying whether all the conditional GUI listeners are statically analyzed to detect commands and *Blob listeners*.
- *Check commands.* We analyzed the conditional statements of GUI listeners to check whether they encompass commands. Then, *recall* measures the percentage of relevant commands that are detected (Equation (4.1)). *Precision* measures the percentage of detected commands that are relevant (Equation (4.2)).

$$Recall_{cmd}(\%) = \frac{|\{RelevantCmds\} \cap \{DetectedCmds\}|}{|\{RelevantCmds\}|} \times 100 \qquad (4.1)$$

$$Precision_{cmd}(\%) = \frac{|\{RelevantCmds\} \cap \{DetectedCmds\}|}{|\{DetectedCmds\}|} \times 100 \qquad (4.2)$$

*RelevantCmds* corresponds to all the commands defined in GUI listeners, *i.e.*, the commands that should be detected by `InspectorGuidget`. *Recall* and *precision* are calculated over the number of false positives (FP) and false negatives (FN). A command incorrectly detected by `InspectorGuidget` while it is not a command, is classified as false positive. A false negative is a command not detected by `InspectorGuidget`.

- *Check Blob listeners.* To check whether a GUI listener is a *Blob listener*, we stated whether the commands it contains concern several widgets. We use the same metrics of commands detection to measure the accuracy of *Blob listeners* detection:

$$Recall_{blob}(\%) = \frac{|\{RelevantBlobs\} \cap \{DetectedBlobs\}|}{|\{RelevantBlobs\}|} \times 100 \qquad (4.3)$$

$$Precision_{blob}(\%) = \frac{|\{RelevantBlobs\} \cap \{DetectedBlobs\}|}{|\{DetectedBlobs\}|} \times 100 \qquad (4.4)$$

Relevant *Blob listeners* are all the GUI listeners that handle more than one command (see Section 4.4). Detecting *Blob listeners* is therefore dependent on the commands detection accuracy.

### 4.5.3 Results and Analysis

In this section, we present the results concerning the evaluated *Blob listeners* and the ability of `InspectorGuidget` in detecting them accurately. The discussion of the findings follows the three research questions we previously introduced.

**RQ1: Command Detection Accuracy.**

Table 4.4 shows the number of commands successfully detected per software system. *TripleA* has presented the highest number of GUI listeners (580), conditional GUI listeners (174), and commands (152). One can notice that despite the low number of conditional GUI listeners that has *TerpPaint* (4), this software system has 34 detected commands. So, according to the sample we studied, the number of commands does not seem to be correlated to the number of conditional GUI listeners.

Table 4.4 also reports the number of FN and FP commands, and the values of the *recall* and *precision* metrics. *TripleA* and *Jmol* revealed the highest number of FN, whereas *TerpPaint* presented the lowest number of FN. The *precision* of the command detection is 99.10 %. Most of the commands (439/443) detected by our algorithm are relevant. We, however, noticed 76 relevant commands not detected leading to an average

Table 4.4: Command Detection Results

| **Software System** | **Successfully Detected Commands (#)** | **FN (#)** | **FP (#)** | **Recall$_{cmd}$ (%)** | **Precision$_{cmd}$ (%)** |
|---|---|---|---|---|---|
| FastPhotoTagger | 32 | 4 | 0 | 88.89 | 100.00 |
| GanttProject | 19 | 6 | 0 | 76.00 | 100.00 |
| JaxoDraw | 99 | 3 | 2 | 97.06 | 98.02 |
| Jmol | 103 | 18 | 2 | 85.12 | 98.10 |
| TerpPaint | 34 | 1 | 0 | 97.14 | 100.00 |
| TripleA | 152 | 44 | 0 | 77.55 | 100.00 |
| OVERALL | 439 | 76 | 4 | 86.05 | 99.10 |

*recall* of 86.05 %. Thus, our algorithm is less accurate in detecting all the commands than in detecting the relevant ones. For example, *TripleA* revealed 44 FN commands and no false positive result, leading to a recall of 77.55 % and a precision of 100 %. The four FP commands has been observed in *JAxoDraw* (2) and *Jmol* (2), leading to a precision of 98.02 % and 98.10 % respectively.

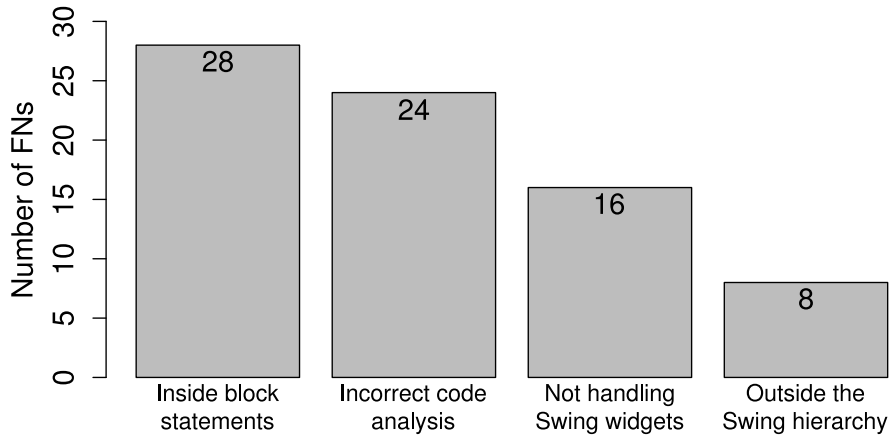


Figure 4.3: Distribution of the false negative commands

Figure 4.3 classifies the 76 FN commands according to the cause of their non-detection. 28 commands were not detected because of the use of widgets *inside block statements* rather than inside the conditional statements our approach analyzes. For example, their conditional expressions refer to boolean or integer types rather than widget or event types. 16 other commands were not detected since they rely on *ad hoc* widgets or GUI listeners. These widgets are developed for a specific purpose and rely on specific user interactions and complex data representation, as we explained in Chapter 3. Thus, our approach cannot identify widgets that are not developed under Java Swing toolkit. All the FN commands reported in this category concern *TripleA* (14) and *Jmol* (2) that use several *ad hoc* widgets. Similarly, we found eight FN commands that use classes defined *outside the Swing class hierarchy*. A typical example is the use of widgets' models (*e.g.*, classes *ButtonModel* or *TableModel*) in GUI listeners.

Also, we identified 24 FN commands caused by an *incorrect code analysis* (either bugs in `InspectorGuidget` or in the *Spoon* library). This result was mainly affected by *Jmol*, that has a listener with 14 commands not detected.

To conclude on RQ1, our approach is efficient for detecting GUI commands that compose GUI listener, even if some improvements are possible.

**RQ2: *Blob listeners* Detection Accuracy.**

Table 4.5 gives an overview of the results of the *Blob listeners* detection per software system. The highest numbers of detected *Blob listeners* concern *TripleA* (22), *Jmol* (18), and *JAxoDraw* (16). Similarly to the command detection, we did not observe a correlation between the number of conditional GUI listeners, commands, and *Blob listeners*. Indeed, *FastPhotoTagger*, *GanttProject*, and *TerpPaint* have presented a quite similar number of detected *Blob listeners* against a different number of conditional GUI listeners.

Table 4.5: *Blob listeners* Detection Results

| **Software System** | **Successfully Detected *Blob listeners* (#)** | **FN (#)** | **FP (#)** | **Recall$_{blob}$ (%)** | **Precision$_{blob}$ (%)** | **Time (ms)** |
|---|---|---|---|---|---|---|
| FastPhotoTagger | 4 | 0 | 1 | 100.00 | 80.00 | 3445 |
| GanttProject | 3 | 0 | 3 | 100.00 | 50.00 | 1910 |
| JaxoDraw | 16 | 0 | 3 | 100.00 | 84.21 | 13143 |
| Jmol | 18 | 1 | 4 | 94.74 | 81.82 | 16904 |
| TerpPaint | 4 | 0 | 0 | 100.00 | 100.00 | 12723 |
| TripleA | 22 | 0 | 19 | 100.00 | 53.66 | 16732 |
| OVERALL | 67 | 1 | 30 | 98.81 | 69.07 | 10810 |

The average recall is 98.81 %. Only the analysis of *Jmol* produced one FN *Blob listener*. In contrast to the command detection, we noticed a higher number of FPs (30). *TripleA* presented the highest number of FP *Blob listeners* (19) followed by *Jmol* (4). The average precision is 69.07 %. The average time spent to analyze the software systems is 10810 milliseconds. It includes the time that *Spoon* takes to process all classes plus the time to detect GUI commands and *Blob listeners*. The worst-case is measured in *TripleA*, *i.e.*, the largest system, with 16732 milliseconds. *Spoon* takes a significant time to load the classes for large software systems (*e.g.*, 12437 milliseconds out of 16732 milliseconds in *TripleA*).

Figure 4.4 depicts the FP *Blob listeners* distribution. We identified 14 FP *Blob listeners* are composed of commands based on checking the *states of widgets*. For instance, two commands can rely on the selection or the non-selection of a checkbox. 13 FP *Blob listeners* with several commands run *over the same widget'* attribute. Two FP *Blob listeners* concern *reverse commands*, *e.g.*, undo operations. One FP *Blob listener* concerns the *state of user interactions*. Commands are in this case detected, while the GUI listener code just updates the running user interaction.

Listing 4.8 gives an example of a FP *Blob listener* detected in *TripleA*. In this example, our algorithm detects two commands (Lines 6–7 and Lines 8–9). This

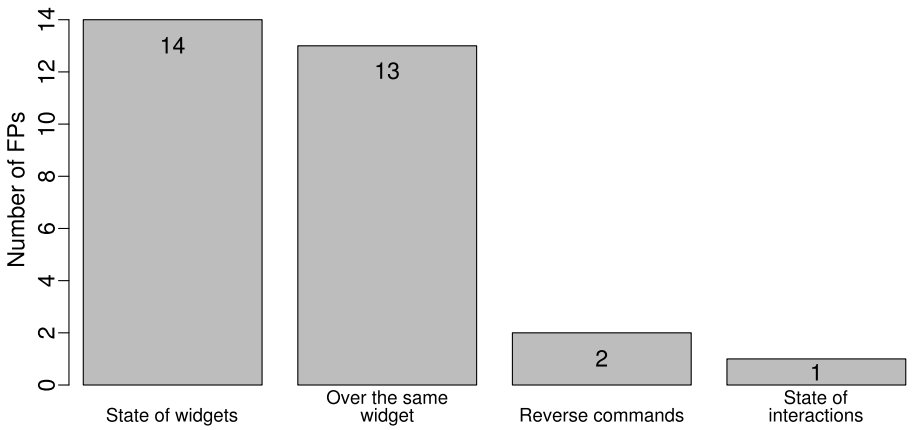


Figure 4.4: Distribution of the false positive *Blob listeners*

GUI listener is therefore classified as a *Blob listener*. However, these two commands are similar since: *i)* they are executed over the same widget attribute (*i.e.*, *m_model* and *m_playerNameLabel*); *ii)* they are reverse - *enablePlayer vs disablePlayer*; and *iii)* the conditional statement handles the state of the widget "*check box*" (*i.e.*, *m_enabledCheckBox* is selected or not). Thus, this GUI listener should not be detected as a *Blob listener*.

```
1  private final ActionListener
2     //GUI listener registration
3    m_disablePlayerActionListener = new ActionListener(){
4      //FP BlobListener
5      public void actionPerformed(final ActionEvent e){
6       if (m_enabledCheckBox.isSelected())//Command #1
7         m_model.enablePlayer(m_playerNameLabel.getText());
8       else//Command #2
9        m_model.disablePlayer(m_playerNameLabel.getText());
10      setWidgetActivation(true);
11     }
12   };
```

Listing 4.8: GUI code excerpt, from *TripleA*[9]

Most of the FP *Blob listeners* have commands contained throughout a single *if-then-else* statement (*e.g.*, Lines 6–9). We observed that these statements are mainly used to handle the state of widgets. For example, checking whether a *check box* is selected (*e.g.*, Line 6) or deselected (*e.g.*, Line 8). Also, most of FP *Blob listeners* identified in *TripleA* are found during a listener registration of a widget (*e.g.*, Line 3). Defining listeners during a widget registration, *i.e.*, as anonymous class, is a good practice (see Section 4.7.2) to avoid *Blob listeners* since we implement the corresponding listener

[9]Path: src/games/strategy/engine/framework/startup/ui/ClientSetupPanel.java; lines: 316-326.

methods for a single widget. We identified a listener in TerPaint with 17 commands[10] running over a registration of an *ad hoc* widget (*e.g.*, canvas).

**RQ3: Do all *Blob listeners* look alike?**

To answer RQ3 we analyze and discuss the Number of Commands per *Blob listener* (NCB) for each software system. Figure 4.5 depicts this distribution.

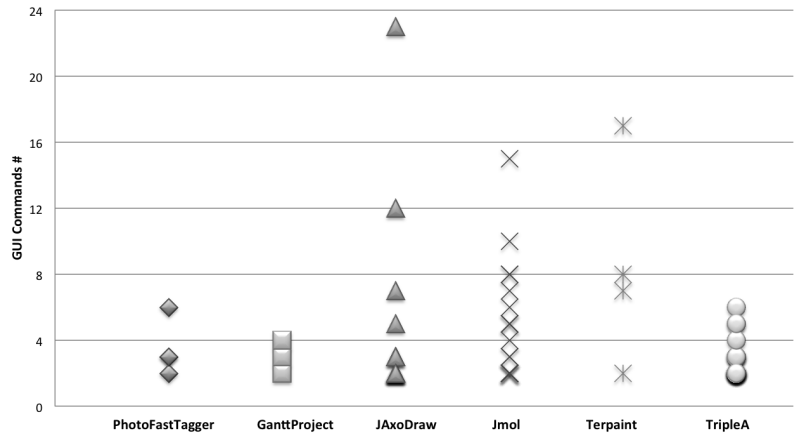


Figure 4.5: Number of GUI commands per *Blob listener* per software system

This chart highlights that NCB can vary in the same software system. For instance, the NCB for *JAxoDraw* ranges from 2 to 23. For *TerpPaint*, it ranges from 2 to 17. By contrast, the NCB for *GanttProject* and *TripleA* ranges from 2 to 4 and 2 to 6, respectively.

In the case of JAxoDraw, the *Blob listener* detected with 23 commands handles events when widgets change their state. Listing 4.9 illustrates the code excerpt from this *Blob listener*. This example shows that for each widget, grouped in a "*edit panel*", that changes its value, the *Blob listener* implements a GUI command. One may note that using a single controller for all the widgets produces a "*long method*" bad smell with all its consequences. Indeed, a Java method with more than a dozen lines is a strong indicator that it is a long method[11]. Besides, this *Blob listener* contains several duplicated code blocks. For example, the Lines 6, 9, and 12 are the same for the commands #1, #2 and #3, respectively. These code lines get the information from the model, for instance, the spinners sequence of values. Duplicated code has negative effects on the maintenance and evolution of software systems [SYA+13]. One way to eliminate those lines is to move the duplicated code to the outside of conditional statements since they handle the same type of a widget (*i.e.*, JSpinner). The duplicated code must be placed before the conditional statements for instance before the Line 5.

[10]Path: src/TerpPaint.java; lines: 4951-5275.
[11]http://martinfowler.com/bliki/CodeSmell.html

The *Blob listener* with 17 commands in *TerPaint* has 125 LOC. The design solution for this *Blob listener* is quite similar to the *Blob listener* in *JAxoDraw*. However, its design solution is more difficult to understand since it has multiple and nested conditional statements. We have observed several conditional statements used to identify the widgets that produced the event (Lines 5, 8, and 11). These conditional statements can be eliminated by following the good coding practices in GUI listeners that we present in Section 4.7.2.

```
1  public void stateChanged(final ChangeEvent evt) {
2          final Component component = (Component) evt.getSource();
3          final String name = component.getName();
4
5    if ("spxin".equals(name)) {//Command #1
6      final SpinnerNumberModel model = component.getModel();
7      setParameter("xPosition", (Integer) model.getNumber());
8    }else if ("spyin".equals(name)) {//Command #2
9      final SpinnerNumberModel model = component.getModel();
10     setParameter("yPosition", (Integer) model.getNumber());
11   }else if ("spx".equals(name)) {//Command #3
12     final SpinnerNumberModel model = component.getModel();
13     setParameter("x", (Integer) model.getNumber());
14   }//... more 20 commands
15 }
```

Listing 4.9: GUI code excerpt, from *JAxoDraw*[12]

The *Blob listeners* detected in *TripleA* and *GanttProject* seem less badly designed. The average NCB for both systems is less than or equal three commands. Indeed, *TripleA* implements several GUI listeners as *anonymous class*, which is one example of the good coding practices in GUI listeners.

We have also observed *Blob listeners* with few commands but several LOCs. For instance, *Blob listeners* detected with three commands in *TripleA* have LOC that range from 23 LOC to 66 LOC whereas in *GanttProject* one *Blob listener* with the same number of commands has 35 LOC. These results highlight that the LOC size of *Blob listeners* detected with the same NCB varies even in a same software system.

## 4.6 Threats to validity

The validity of the study to identify and detect automatically the *Blob listeners* is considered from two perspectives as follows:

**External validity.** This threat concerns the possibility to generalize our findings. We designed the experiments using six Java Swing open-source software systems to diversify the observations. These six unrelated software systems are developed by different persons and cover various user interactions. Several selected software systems have been used in previous research work, *e.g.*, *GanttProject* [FBZ12, AZJH11], *Jmol* [AZJH11], and *TerpPaint* [CHM12] that have been extensively used against GUI testing tools.

[12]Path: src/net/sf/jaxodraw/gui/panel/edit/JaxoOptionsPanelListener.java; lines: 83-183.

Our implementation and the initial study (Section 4.2) focus on the Java Swing toolkit only. We focus on the Java Swing toolkit because of its popularity and the large quantity of Java Swing legacy code. We provide on the companion web page examples of *Blob listeners* in other Java GUI toolkits, namely GWT, SWT, and JavaFX[1].

**Construct validity.** This threat relates to the perceived overall validity of the experiments. The detection of FNs and FPs have required a manual analysis of all the GUI listeners of the software systems. To limit errors during this manual analysis, we added a debugging feature in `InspectorGuidget` for highlighting GUI listeners in the code. We used this feature to browse all the GUI listeners and identify their commands to state whether these listeners are *Blob listeners*. During our manual analysis, we did not notice any error in the GUI listener detection. Another threat is the fact that we manually determined whether a listener is a *Blob listener*. To reduce this threat, we carefully inspected each GUI command highlighted in our tool.

## 4.7 Discussion

In the next two subsections, we discuss the scope of `InspectorGuidget` and good practices that should be used to limit *Blob listeners*.

### 4.7.1 Scope of the Approach

Our approach has the following limitations. First, `InspectorGuidget` currently focuses on GUIs developed using the Java *Swing* toolkit. This is a design decision since we leverage Spoon, *i.e.*, a library to analyze *Java* source code. However, our solution is generic and can be applied to cover other GUI toolkits.

Second, our solution is limited to analyze GUI listeners and associated class attributes. We identified several GUI listeners that dispatch the event processing to methods, as illustrated in Listing 4.10. The GUI listener (Lines 1–3) was not analyzed by our algorithm since it was not identified as a conditional GUI listener. However, the method invoked in Line 2 contains a strong candidate for a *Blob listener* (Lines 4–16). Our implemented static analyses can be extended to traverse these methods to improve its performance. To identify such methods as listeners, our algorithm must analyze each listener statement to: 1. identify all call methods in a listener; and 2. analyse such methods to infer if they share the same parameter and return type (*e.g.*, void in Java Swing listeners) of that listener.

Last, the criteria for the *Blob listeners* detection should be augmented by inferring the related commands. An example of related commands are a command and its reverse one (the undo operation). For example, when a GUI listener is a *Blob listener* candidate, our algorithm should analyze its commands by comparing their commonalities (*e.g.*, shared widgets and methods). The goal is to detect commands that form in fact a single command. Considering the GUI listener presented in Listing 4.8 and the proposed detection, the algorithm will not detect this GUI listener as *Blob listener* since the commands #1 and #2 will be marked as related commands.

```
1  public void actionPerformed(ActionEvent evt) {
2    RepeatActionPerformed(evt);
3  }//...
4  public void RepeatActionPerformed(ActionEvent evt){
5    if (curButton == select){//Command #1
6      toolSelect.deSelect(center);
7    }else if (curButton == magicSelect) {
8      toolMagicSelect.deSelect(center);//Command #2
9    }else if (curButton == selectall) {
10     toolSelectall.deSelect(center);//Command #3
11   }else if (curButton == curve) {
12     toolCurve.deSelect(center);//Command #4
13   }else if (curButton == polygon) {
14     toolPolygon.deSelect(center);//Command #5
15   } //three others ifs are implemented
16 }
```

Listing 4.10: GUI code excerpt, from *TerpPaint*[13]

### 4.7.2 Good Practice

The analysis of Java Swing GUI listeners having a low CC reveals (see Section 4.2) what we consider as a good practice for developing GUI listeners. In most of the cases, this good practice consists of producing one command *per* listener (*i.e.*, managing *one* widget per listener). In some particular cases, several widgets can be managed by a single listener when a user interaction involves multiple widgets (*e.g.*, a drag-and-drop of an item from a list to another list). Because of features introduced in different Java versions, variants of this good practice exist as explained as follows.

**Listeners as anonymous classes.** Listing 4.11 is an example of this good practice. A listener, defined as an anonymous class (Lines 3–6), registers with one widget (Line 2). The methods of this listener are then implemented to define the command to perform when an event occurs. Since such listeners have to handle only one widget, *if* statements used to identify the involved widget are not more used, simplifying therefore the code.

```
1  private void registerWidgetHandlers() {
2   view.resetPageButton().addActionListener(
3     new ActionListener() {
4     @Override public void actionPerformed(ActionEvent e) {
5       requestData(pageSize, null);}
6   });
7   view.previousPageButton().addActionListener(
8     new ActionListener() {
9       @Override public void actionPerformed(ActionEvent e) {
10        if(hasPreviousBookmark())
11          requestData(pageSize, getPreviousBookmark());}
12    });//...
13 }
```

Listing 4.11: Good practice for defining controllers: one widget per listener

[13]Path: src/TerpPaint.java; lines: 1125-1127;4525-4557.

**Listeners as lambdas.** Listing 4.12 illustrates the same code than Listing 4.11 but using Lambdas supported since Java 8. Lambdas simplify the implementation of anonymous class that have a single method to implement. With Lambdas, the header declaration of an anonymous class and of its unique method are not more required. The parameters of the unique method (here *event*, Line 10) are declared followed by an arrow pointing to instructions composing the body of the Lambdas (*i.e.*, of the method). Note that only listeners composed of a single method can be written using Lambdas.

```
1  private void registerWidgetHandlers() {
2   view.resetPageButton().addActionListener(
3                e -> requestData(pageSize, null));
4
5   view.previousPageButton().addActionListener(e -> {
6    if (hasPreviousBookmark())
7      requestData(pageSize, getPreviousBookmark());
8   });
9  //...
10 }
```

Listing 4.12: Same code than in Listing 4.11 but using Java 8 Lambdas

**Listeners as classes.** In some cases, listeners have to manage different intertwined methods. This case notably appears when developers want to combine several listeners or methods of a single listener to develop a more complex user interaction. For example, Listing 4.13 is a code excerpt that describes a mouse listener where different methods are managed: *mouseClicked* (Line 3), *mouseReleased* (Line 8), and *mouseEntered* (Line 11). Data are shared among these methods (*isDrag*, Lines 4 and 9).

```
1  class IconPaneMouseListener implements MouseListener {
2    // ...
3    @Override public void mouseClicked(MouseEvent e) {
4      if(!isDrag) {
5        // ...
6      }
7    }
8    @Override public void mouseReleased(MouseEvent e) {
9      isDrag = false;
10   }
11   @Override public void mouseEntered(MouseEvent e) {
12     isMouseExited = false;
13     // ...
14   }
15 }
```

Listing 4.13: A GUI listener defined as a class

## 4.8 Towards a refactoring

A next step of our static analysis approach is to implement a behavior-preserving algorithm to automatically refactor the detected *Blob listeners*. In this section, we present

the next steps towards a refactoring *Blob listeners*. We have started the implementation of this algorithm. The global solution consists of implementing three main steps:

1. Detecting *Blob listeners*;
2. Locating where the GUI commands included in *Blob listeners* are registered; and
3. Extracting GUI commands by producing one command per listener, for instance, commands into listeners as anonymous classes.

The step one is completed and evaluated as we detailed in Section 4.4 and Section 4.5. During the evaluation of *Blob listeners* detection we identified some criteria to precise our static analysis algorithm and thus decrease the number of false positive *Blob listeners* (recall Section 4.7).

We have implemented the step two based on the good coding practices that we explained in Section 4.7.2. Figure 4.6 shows the activity diagram of the algorithm to locate in the source code where the widget involved in a command is registered. First, the algorithm detects all listener registrations in the source code (*e.g.*, *buttonUndo.addActionListener*). Also, the listener methods that implement a listener interface targeted by a registration are recovered. This information will be used to check whether one of these methods matches the listener method where a command is found. This preliminary check improves the overall performance of our algorithm. Next, the registrations are analyzed to obtain the widget targeted by them. Once the widget is identified, the algorithm obtains the statements referring to this widget. These statements contain the information about the widget properties such as *buttonUndo.setActionCommand("undo")*. A similar process is performed to extract the properties of commands. The difference is that the properties of commands are extracted from their conditionals statements (*e.g.*, *e.getActionCommand.equals("undo")*). Finally, the properties of commands and widgets are compared, if they match we relate a command with a registration.

Note that gathering all the properties of a widget or a command is more accurate. For example, some conditional expressions do not refer directly to a widget attribute: *e.getActionCommand().equals("undo")*, where *e* refers to a non-specified widget. Instead, the widget attribute *buttonUndo* is easily identified in the conditional expression *e.getSource() == buttonUndo*.

The last step consist of extracting a command to its registration (in the case we choose to refactor a *Blob listener* as as anonymous class). However, several factors must be analysed to perform a safe refactoring to preserve the code behavior. For example, the dependency between the listeners as well as the attributes shared by them. Also, a GUI can be developed in different ways, for instance, a listener may register on a component, which represents several widgets. In such a case, there is no single widget targeted by a registration (*e.g.*, *this.addEventListener*).

While the goal of the second step is to locate the corresponding listener registrations of commands, we observed that this step can also be used to identify some GUI faults. For example, Listing 4.3 illustrates a *Blob listener* with GUI faults in their two commands (Command #1: Lines 12–18 and Command #2: Lines 19–21). In this case, our

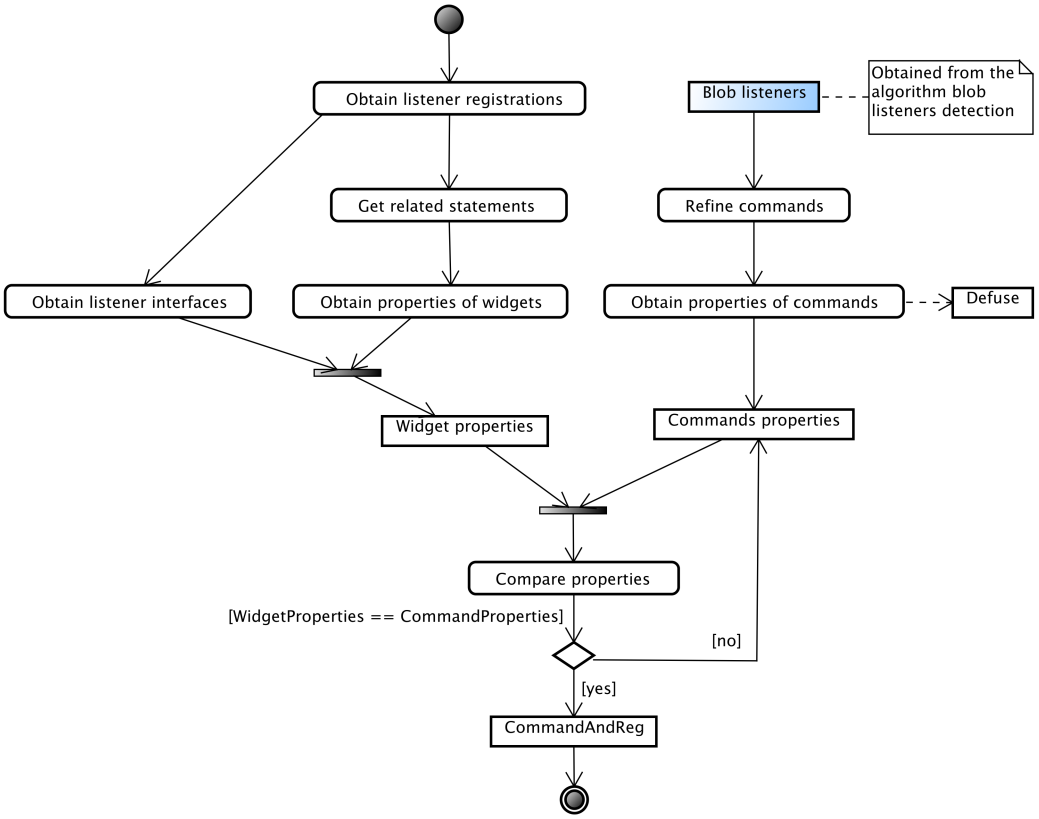


Figure 4.6: The activity diagram of the algorithm to detect the command registration

algorithm will not identify the listener registrations for both commands since the GUI faults cause the mismatch between the properties. The values of the property invoked by the commands (*e.g.*, label "*Clear*" in Command #1) to identify the widget that produced the event differ from the property values (*e.g.*, label "*Clean*") of the widgets targetted by the listener registration (*e.g.*, clearButton in Line 5).

## 4.9 Conclusion

We identified and characterized a *Blob listener* as a new design smell specific for GUIs. The empirical study has shown evidences that implemented Java GUI listeners use more conditional statements than other methods. We pored over these listeners and identified the *Blob listeners* can degrade the aspects of GUI quality code for instance by leading the introduction of GUI faults. We have observed three variations of a *Blob listener* that may occur in Java Swing listeners. Our proposed static analysis, implemented in `InspectorGuidget`, has successfully detected 67 *Blob listeners* out of 68 in six Java systems.

We believe that the presence of *Blob listeners* affects the quality of GUI code. We

intend to investigate whether some GUI faults are accentuated by GUI design smells. This study can be assisted by our GUI fault model presented in Chapter 3. The effects of GUI design smells on GUI faults can help software engineers to prevent specific GUI faults, for instance by addressing the research question: *which kind of GUI faults found in GUI controllers are correlated to the presence of Blob listeners?* Also, this study will help us to demonstrate the empirical evidence that an automated refactoring of *Blob listeners* is worth to be done.

# Part III

# Conclusion and Perspectives

# Chapter 5

# Conclusion and Perspectives

This chapter summarizes the contributions of this thesis in GUI testing domain, and outlines the directions for future research.

## 5.1 Summary of contributions

From a software engineering point of view, this thesis contributes to improve the quality and reliability of interactive systems, *i.e.*, software systems that provide users with user interfaces (graphical user interfaces in our case). The contributions are summarized below.

**GUI fault model.** A GUI fault model is proposed to describe a set of GUI faults. This model is an *original GUI fault classification scheme* because of the following reasons. First, the fault model describes GUI faults that stem from *standard GUIs* (*i.e.*, WIMP GUIs) and *advanced GUIs* (*i.e.*, post-WIMP GUIs). Second, the fault model is *structured at two fault levels*: user interface and user interaction faults. Such a structure is essential to distinguish faults that affect the graphical components that compose GUIs from the ones that affect the user interactions that occur between users and systems. Third, our fault model covers seminal HCI concepts, mandatory to identify GUI faults that concern complex user interactions. Last, the fault model presents in detail *which* GUI is affected and *how* it occurs as a GUI failure.

The coverage and the benefits of the proposed fault model have been demonstrated as follows.

- *An empirical study of real GUI failures*: 279 GUI-related bug reports of five highly interactive open-source GUIs have been successfully classified using our fault model. This study highlights the relevance of our fault model in identifying and classifying faults observed in existing GUIs.

- *Evaluation of two GUI testing tools against real GUI failures*: some failures classified in our fault model that concern standard widgets were not detected by two standard GUI testing tools (GUITAR and Jubula). This evaluation demonstrates that some kinds of faults related to WIMP widgets are more difficult to detect

by the GUI testing tools, for instance, GUI faults found in the auto-completion feature. Also, the regression GUI testing tools are not able to detect some failures when a GUI faulty version is used to produce the test cases.

- *Development of GUI mutants*: 65 GUI mutants are derived from our fault model and planted in a highly interactive open-source system - LaTeXDraw (22 into standard widgets and 43 into *ad hoc* widgets). These mutants are freely available and can be reused by developers for benchmarking their GUI testing tools.

- *Evaluation of GUI testing tools against GUI mutants*: 43 out of 65 mutants were not killed by that standard GUI testing tools, where 40 of them concern *ad hoc* widgets and their user interactions. This evaluation demonstrates that GUI testing frameworks fail in detecting several failures that stem from *ad hoc* widgets, their multi-event interactions and data presentation.

- *A precise analysis of a standard GUI testing framework*: an study explains the current limitations of GUI testing frameworks that focus on standard GUIs. This study helps software testers to understand why GUI failures that stem from GUI faults in our fault model were not detected by standard GUI testing tools.

**Automatic detection of a GUI design smell.** A systematic static analysis approach is developed to automatically detect a new type of design smell we identified and characterized, and that affect GUI implementations: the *Blob listener*. This approach has been implemented in a tool publicly available called `InspectorGuidget`. The achievements are outlined below.

- *An empirical analysis of GUI controllers*: the code of Java Swing listeners retrieved from 511 code repositories in Github was observed. The results pointed out a higher use of conditional statements (*e.g.*, *if*) in listeners used to identify widgets.

- *Identification of GUI design smells*: we identified a critical bad coding practice that we characterized as a new type of GUI design smell: *Blob listener*. *Blob listeners* are listeners that handle several GUI commands. A GUI command is a set of statements executed in reaction to a user interaction.

- *A static analysis to automatically detect the presence of Blob listeners in Java systems*: contrary to other code smells that can be located with structural rules on the whole code base of a software system, the detection of *Blob listener* requires a preliminary semantic code analysis to isolate the parts of the code base related to GUI implementation (*e.g.*, identification of GUI listeners and conditional GUI listeners). The second step of our analysis consisted in detecting the GUI commands, *i.e.*, GUI actions, that can produce each implemented GUI listener.

- *An empirical evaluation*: 67 out of 68 *Blob listeners* were successfully detected by `InspectorGuidget` in six highly interactive systems.

- *Good coding practices to avoid the use of Blob listeners*: these practices represent a step towards the automated refactoring of *Blob listeners*.

**Experiments and tools.** We have run our experiments on large interactive systems, which required building tools that can handle the complexity of real-world software.

The subject systems have been selected by using several criteria such as interactive features (*e.g.*, direct manipulation, feedback), input devices (*e.g.*, mouse, touch), the type of widgets (*ad hoc*), data graphics (2D or 3D objects). We also used an open-source interactive system to calibrate our tools during the experiments. For example, LaTeXDraw has been used to develop the GUI mutants as well as to bootstrap `InspectorGuidget` against Java GUI listeners.

The empirical analysis to evaluate the GUI fault model against real GUI failures was performed by carefully selecting the bug reports of five interactive systems. So, we analysed the bug report artifacts (*e.g.*, description, patches, logs, or stack traces) to identify and classify the faults in our fault model. Our examination of GUI development practices relies on open-source code hosting repositories that we identified several interactive systems.

To run the GUI testing tools against the GUI mutants (*e.g.*, GUITAR and Jubula), we have studied their documentation such as developer/user manuals, technical reports or research papers, homepages, and discussion forums. This study allowed us to minimize several threats regarding our experiments. To develop the GUI mutants, we have studied the code of LaTeXDraw in different versions to understand the GUI behavior of Java Swing implementations.

To implement `InspectorGuidget` we leveraged the existing tools to provide a more mature solution. For example, our static analysis uses Spoon to transform and analyse the Java source code. We also have implemented `InspectorGuidget` as an Eclipse plugin to facilitate the integration with other Java tools. Indeed, the evaluation of `InspectorGuidget` has been performed on six Java applications, which differ from the systems that we evaluated our fault model.

We built a complete data set of all experiments performed during this research work. This data set can be used to study for instance the correlation between GUI implementations and their GUI faults.

## 5.2 Perspectives

The proposed fault model has been used to identify the limits of GUI testing tools. We believe that this model can serve as the basis to improve the GUI testing techniques to cover more GUI faults. Also, these faults may be accentuated by the presence of GUI design smells. An example is the two bad coding practices *listener with empty bodies* and *unsafe registration of listeners*, as we explain later, that may introduce faults.

The directions for future research around the contributions of this thesis are presented below.

### GUI Design smells

In this thesis, we have characterized a GUI design smell: *Blob listener*. However, we believe that other GUI design smells may be identified from bad development practices. In Chapter 2, we have presented *bug finders* such as Findbugs and PMD Java tools. While these tools are widely used to detect bad programming practices and bugs in the Java source code, they do not focus on coding problems that affect the GUI source code. As a future work, we envision a set of checking rules, embedded in a tool similar to PMD, which automatically check for potential defects in GUI code at every project build. As an initial step in this direction, we have implemented in `InspectorGuidget` the automatic detection of simple bad coding practices that affect Java Swing code. The mere examples are presented below.

One bad practice we observed is the *unsafe registration of listeners*[1]. This problem occurs when a listener is registered in its constructor, for instance, by the code line "*buttonA.registerListener(this)*" in Listing 5.1 (Line 6). In such a case, the reference *this* is exposed to another thread before the constructor is finished.

```
1  public class Listener implements ActionListener{
2
3    public Listener(ActionEvent event){
4      // Do the initialization
5      //Register the listener on a buttonA
6         buttonA.registerListener(this);
7    }
8    public void actionPerformed(ActionEvent event){
9      Object source = event.getSource();
10     if(source == buttonA){
11       //Manage the event from buttonA
12     }
13    }
14 }
```

Listing 5.1: Example of unsafety listener registration

A failure may occur when that listener is used in its subclasses leading to a race condition. The following code depicts this case. When a subclass extends the class *Listener*, the call "super" must be the first statement in its constructor. In this case, the event listener (*i.e.*, *SubClass.actionPerformed()*) could get called while the subclass fields (*e.g.*, list) are not initialized yet.

```
1  public class SubClass extends Listener{
2    public SubClass (ActionEvent event){
3      super(event);
4      list = Collections.synchronizedList(new ArrayList<ActionEvent>());
5      //...
6    }
7    public void actionPerformed(ActionEvent e){
8      list.add(e);
9      //...
10   }
11 }
```

[1] http://www.ibm.com/developerworks/library/j-jtp0618/

Another bad practice is several *listeners with empty bodies*. This practice is common in listener interfaces that require more than one method to be implemented (*e.g.*, *MouseListener* interface requires five listener methods). This kind of methods allows empty bodies. However, a code with several empty method bodies is harder to read and maintain[2]. Furthermore, such flexibility may introduce unintended faults. For example, a listener is registered on a widget and none of its listener methods are implemented (*e.g.*, all their bodies are empty or commented). Listing 5.2 gives an example of this practice observed in Jmol. All the three methods required by the *MenuListener* interface are not implemented: empty body (Line 10 and Line 12) or commented body (Line 7). In this case, an error may occur when a user interacts with that menu (*i.e.*, processingMenu in Line 4). PMD detects unused private methods[3], *i.e.*, private methods with empty body, without any comment in their body. This rule, however, will not detect Java listeners with empty body since they are public methods.

```
1  processingMenu = new JMenu();
2  processingMenu.setMnemonic('P');
3  processingMenu.setText("Processing");
4  processingMenu.addMenuListener(new MenuListener() {
5   @Override
6   public void menuSelected(MenuEvent e) {
7     //jsvpPopupMenu.setEnables(mainFrame.viewer.selectedPanel);
8   }
9   @Override
10  public void menuDeselected(MenuEvent e) {}
11  @Override
12  public void menuCanceled(MenuEvent e) {}
13 });
```

Listing 5.2: GUI code excerpt, from Jmol[4]

### Domain-specific mutants

A GUI code can by affected by different kinds of faults that we presented in our fault model. We have developed GUI mutants derived from our fault model using domain-independent mutation operators. GUI mutant operators, however, are domain-specific and thus may differ from one GUI toolkit to another. To develop Java Swing mutants, we have studied the behavior of GUI implementations in Java code. Thus, we designed manually the mapping between Java Swing mutants and the GUI faults in our fault model. For example, the creation of a standard and interactive widget in Java Swing is done by three basic operations:

1. instantiate a widget and add it to a GUI;
2. register an event to this widget; and
3. provide the implementation for its listener.

[2] https://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html

[3] http://pmd.sourceforge.net/pmd-4.3.0/xref/net/sourceforge/pmd/rules/UnusedPrivateMethodRule.html

[4] Path: src/jspecview/application/ApplicationMenu.java; lines: 391-407.

Given this behavior, GUI faults can be forged by breaking the code involved in these operations.

An example is the mutation operator introduced in the widget registration. This mutant changes the statement responsible for registering the listener on a widget. In such a case, when a user interacts with that widget the desired action is not executed Furthermore, some faults are more difficult to be modelled. One example is to forge a fault into the drag-and-drop (DnD) interaction. The basic behavior of a DnD starts by a *mouse pressure* followed by at least one *move* and ends with a *release*. In Java Swing, a DnD can be implemented by three GUI listeners which are *mousePressed*, *mouseDragged*, and *mouseReleased*. A mutant Java operator to introduce the fault "*interaction behavior*" in a DnD may be developed by breaking some attributes shared by these Java Swing listeners. Similarly, the unsafe use of the *this* reference in constructors exemplified previously by the bad practice *unsafe registration of listeners* may be used to forge GUI faults.

We also believe that the creation of GUI mutants may be more particular in other domains, for instance, GUI actions in XML. A future research can provide a mapping between GUI faults and the domain-specific mutants. One challenge is to develop a program analysis to automatically introduce such mutants in GUIs. For instance, the Java Swing mutants that we developed may be automatically planted in other Java Swing GUIs.

#### Refactoring *Blob listeners*

We presented the steps towards the refactoring of *Blob listeners* in Chapter 4. Refactoring *Blob listeners*, however, may involve another substantial cost and risks [KZN12] that we should carefully study. We believe that *Blob listeners* can potentially introduce some GUI faults as we exemplified in Listing 4.3 (recall Chapter 4). In this example, refactoring the *Blob listener* will improve the GUI code quality (*e.g.*, decreasing the cyclomatic complexity) and correct two GUI faults. Listing 5.3 gives a piece of code to demonstrate the refactoring of the *Blob listener* shown in Listing 4.3.

```
1  private void registerWidgetHandlers() {
2   view.clearButton.addActionListener(e -> {
3    if (noChannel){
4      clearPanel();
5      return;
6    }
7    sendClearPanelMsg();
8   });
9   view.leaveButton.addActionListener(
10   e -> stop());
11 //...
12 }
```

Listing 5.3: Refactoring a small *Blob listener* using Java 8 Lambdas[5]

[5]The original code is available here: https://github.com/LinhTran95/Asg7--Team10/blob/master/jWhiteBoard/src/jWhiteBoard/JWhiteBoard.java

We can refactor this *Blob listener* as lambdas supporting by Java 8 (recall 4.7.2 in Chapter 4) since its implementation has a single method, *i.e.*, *actionPerformed*. In this case, the method *actionPerformed* and the identification of widgets that produced the event (used by GUI commands) are not required anymore (see Lines 2–8 and Lines 9–11). Thus, the two reported faults will be removed when the refactoring is done.

A future research on the refactoring of *Blob listeners* should first select which kinds of good practices will be applied and then provide the mechanisms to preserve the code's behavior. For instance, refactoring a *Blob listener* by producing one GUI command per listener (*i.e.*, *listener as anonymous class* recall Chapter 4) should locate where the widget used in a GUI command is registered and handle its dependencies (*e.g.*, shared attributes) within the *Blob listener* to perform the refactoring correctly. Thus, one challenge is to ensure that the refactoring approach will not introduce new GUI faults. To prevent this, the refactored software systems must be tested using non-regression GUI test suites generated from the original software systems. Another challenge is to ensure that the refactoring will not degrade the GUI code quality. So, we should compare the code quality of the original and refactored code by using some code quality metrics such as cyclomatic complexity, code duplication, code cohesion, or the correction of GUI faults in *Blob listeners*.

# Appendix

## Beizer's Taxonomy

Beizer's taxonomy adapted by Brooks *et al.* [BRM09] is presented in Table 5.1.

Table 5.1: Beizer's taxonomy adapted to cover GUIs [BRM09]

| ID | Defect types categories |
|---|---|
| **1xxx** | *Functional Bugs: Requirements and Features* |
| | 11xx Requirements Incorrect |
| | 12xx Logic |
| | 13xx Completeness |
| | 14xx Verifiability |
| | 15xx Presentation |
| | 16xx Requirements Changes |
| **2xxx** | *Functionality As Implemented* |
| | 21xx Correctness |
| | 22xx Completeness – Features |
| | 23xx Completeness – Cases |
| | 24xx Domains |
| | 25xx User Messages and Diagnostics |
| | 26xx Exception Conditions Mishandled |
| **3xxx** | *Structural Defect* |
| | 31xx Control Flow and Sequencing |
| | 32xx Processing |
| **4xxx** | *Data Defect* |
| | 41xx Data Definition, Struc., Declaration |
| | 42xx Data Access and Handling |
| **5xxx** | *Implementation Defect* |
| | 51xx Coding and Typographical |
| | 52xx Standards Violations |
| | 53xx *GUI Defects* |
| | 54xx *Software Documentation* |
| | 55xx *User Documentation* |

| | |
|---|---|
| **6xxx** | *Integration Defect* |
| | 61xx Internal Interfaces |
| | 62xx External Interfaces |
| | 63xx *Configuration Interfaces* |
| **7xxx** | *System and Software Architecture Defect* |
| | 71xx OS |
| | 72xx Software Architecture |
| | 73xx Recovery and Accountability |
| | 74xx Performance |
| | 75xx Incorrect diagnostic |
| | 76xx Partitions and overlays |
| | 77xx Environment |
| | 78xx 3rd Party Software |
| **8xxx** | *Test Definition or Execution Bugs* |
| | 81xx *System Setup* |
| | 82xx Test Design |
| | 83xx Test Execution |
| | 84xx Test Documentation |
| | 85xx Test Case Completeness |

## GUI code examples

```
1  public class CalculatorLayout extends JFrame implements ActionListener{
2     private JButton bOne = new JButton("1");
3     private JButton bTwo = new JButton("2");
4     private JButton bSin = new JButton("sin");
5     private JButton bCos = new JButton("cos");
6     private JTextField tfDisplay = new JTextField();//result displaying screen
7     private JTextField tfRawInput = new JTextField();
8     //... more 41 declarations/instantiation of swing widgets variables
9
10   @Override public void actionPerformed(ActionEvent e) {
11        Object src = e.getSource();
12    if (e.getSource() == bOne){ //Command#1
13      if(operation == '='){
14        sDisplay = "1";
15        sRawInput = "1";
16        tfRawInput.setText(sRawInput);
17        operation = '';
18        }
19      else{
20        sDisplay = sDisplay + "1";
21        sRawInput += "1";
22        tfRawInput.setText(sRawInput);
23      }
24    } //... more 15 GUI commands to handle buttons bTwo, etc.
25    else if (e.getSource() == bEqual && !sDisplay.equals("")){//Command #18
26      number2   = Double.parseDouble(sDisplay);
27        if(operation == '+'){
28          result = number1 + number2;
29        }
30        else if(operation == '-'){
31          result = number1 - number2;
32        }
33        else if(operation == '*') {
34          result = number1 * number2;
35        }//... more 3 "else if" conditional statements
36        String temp = "";
37        if(isPoint  operation == '/'){
38          tfDisplay.setText(""+result);
39          temp = ""+result;
40        }
41        else if(!isPoint){
42          tfDisplay.setText(""+(long)result);
43          temp = ""+(long)result;
44        }
45        sDisplay = "";
46        number1 = result;
47        isPlus = true;
48        isPoint = false;
49        isOperation = true;
50        sRawInput += "=";
51        tfRawInput.setText(sRawInput);
52        sRawInput = temp;
53        operation = '=';
54    }//...more 21 GUI commands to handle buttons bsin, bcos, etc.
55    }
56 }
```

Listing 5.4: GUI code excerpt, from the *Scientific calculator*[6]

[6] https://github.com/alimranahmed/Scientific-Calculator-using-JAVA-swing

# Bibliography


[ABA04] Roger T. Alexander, James M. Bieman, and Anneliese A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical report, Department of Computer Science, Colorado State University, 2004.

[ACP12] Caroline Appert, Olivier Chapuis, and Emmanuel Pietriga. Dwell-and-spring: undo for direct manipulation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'12)*, pages 1957–1966. ACM, 2012.

[AFT+12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of android applications. In *Proc. of ASE'12*, pages 258–261, 2012.

[APB+12] S. Arlt, A. Podelski, C. Bertolini, M. Schaf, I. Banerjee, and A.M. Memon. Lightweight static analysis for GUI testing. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 301–310, 2012.

[AZJH11] Andrea Adamoli, Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Automated GUI performance testing. *Software Quality Journal*, 19(4):801–839, 2011.

[BA06] J.S. Bsekken and R.T. Alexander. A candidate fault model for aspectj pointcuts. In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 169–178, 2006.

[BB10] Arnaud Blouin and Olivier Beaudoux. Improving modularity and usability of interactive systems with Malai. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS'10, pages 115–124, 2010.

[BBG11] F. Belli, M. Beyazit, and N. Güler. Event-based GUI testing and reliability assessment techniques – an experimental insight and preliminary results. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 212–221, 2011.

[BC06] Jon Arvid Børretzen and Reidar Conradi. Results and experiences from an empirical study of fault reports in industrial projects. In *Proc. of PROFES'06*, pages 389–394, Berlin, Heidelberg, 2006. Springer-Verlag.

[BCB13] Olivier Beaudoux, Mickael Clavreul, and Arnaud Blouin. Binding orthogonal views for user interface design. In *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, page 4. ACM, 2013.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Co., 1990.

[Bin96] Robert V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6(3-4):125–252, 1996.

[Bin99] Robert V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BL00] Michel Beaudouin-Lafon. Instrumental interaction: An interaction model for designing post-wimp user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 446–453, New York, NY, USA, 2000. ACM.

[BL04] Michel Beaudouin-Lafon. Designing interaction, not interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 15–22, New York, NY, USA, 2004. ACM.

[BLL00] Michel Beaudouin-Lafon and Henry Michael Lassen. The architecture and implementation of cpn2000, a post-wimp graphical application. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, UIST '00, pages 181–190, New York, NY, USA, 2000. ACM.

[Blo09] Arnaud Blouin. *Un modèle pour l'ingénierie des systèmes interactifs dédiés à la manipulation de données*. PhD thesis, University of Angers, France, 2009.

[BLS05] Mike Barnett, K.RustanM. Leino, and Wolfram Schulte. The spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin Heidelberg, 2005.

[BM11] Rex Black and Jamie L. Mitchell. *Advanced Software Testing - Vol. 3: Guide to the ISTQB Advanced Certification As an Advanced Technical Test Analyst*. Rocky Nook, 1st edition, 2011.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[BMN+11] Arnaud Blouin, Brice Morin, Grégory Nain, Olivier Beaudoux, Patrick Albers, and Jean-Marc Jézéquel. Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 85–94, 2011.

[BRM09] P. Brooks, B. Robinson, and A.M. Memon. An initial characterization of industrial graphical user interface systems. In *Proc. of ICST'09*, 2009.

[CBC+92] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and M-Y Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Trans. Softw. Eng.*, 18(11):943–956, 1992.

[CHM12] M.B. Cohen, Si Huang, and A.M. Memon. AutoInSpec: Using missing test coverage to improve specifications in GUIs. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 251–260, 2012.

[CTR05] M. Ceccato, P. Tonella, and F. Ricca. s aop code easier or harder to test than oop code? In *Proceedings of the 1st Workshop on Testing Aspect-Oriented Programs (WTAOP), held in conjunction with the 4th international Conference on Aspect-Oriented Software Development (AOSD)*, 2005.

[CW12] Woei-Kae Chen and Jung-Chi Wang. Bad smells and refactoring methods for GUI test scripts. In *Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, pages 289–294, 2012.

[DJK+99] Siddhartha R Dalal, Ashish Jain, Nachimuthu Karunanithi, JM Leaton, Christopher M Lott, Gardner C Patton, and Bruce M Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294. ACM, 1999.

[DMM05] A. Deursen, A.M. Marin, and L.M.F. Moonen. *A Systematic Aspect-oriented Refactoring and Testing Strategy, and Its Application to JHotDraw*. Report SEN. CWI, Centrum voor Wiskunde en Informatica, 2005.

[FBZ12] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):5:1–38, 2012.

[FFP+13] Camille Fayollas, Jean-Charles Fabre, Philippe A. Palanque, Eric Barboni, David Navarre, and Yannick Deleris. Interactive cockpits as critical applications: a model-based and a fault-tolerant approach. *IJCCBS*, 4(3):202–226, 2013.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GPEM09] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 255–258. IEEE, 2009.

[Gra92] Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[GSS15] Ganesh Samarthyam Girish Suryanarayana and Tushar Sharma. *Refactoring for Software Design Smells*. Morgan Kaufmann (Elsevier), Boston, 2015.

[GXF09] M. Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 408–418, 2009.

[GZDG15] Maria Anna G. Gaitani, Vassilis E. Zafeiris, N.A. Diamantidis, and E.A. Giakoumakis. Automated refactoring to the null object design pattern. *Information and Software Technology*, 59(0):33–52, 2015.

[HHN85] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. *Hum.-Comput. Interact.*, 1(4):311–338, 1985.

[HZBS14] Tracy Hall, Min Zhang, David Bowes, and Yi Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology*, 23(4):33:1–33:39, September 2014.

[IEC95] IEC. Nuclear power plants - main control-room - verification and validation of design, 1995.

[IEC10] IEC. Nuclear power plants - control rooms - design, 2010.

[IEE10] IEEE. Standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of. IEEE Std 1044-1993), New York*, 2010.

[JGH+08] Robert J.K. Jacob, Audrey Girouard, Leanne M. Hirshfield, Michael S. Horn, Orit Shaer, Erin Treacy Solovey, and Jamie Zigelbaum. Reality-based interaction: A framework for post-wimp interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 201–210, New York, NY, USA, 2008. ACM.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011.

[JS07] Alejandro Jaimes and Nicu Sebe. Multimodal human-computer interaction: A survey. *Comput. Vis. Image Underst.*, 108(1-2):116–134, 2007. "Special Issue on Vision for Human-Computer Interaction".

[Kar08] Alexandros Karagkasidis. Developing GUI applications: Architectural patterns revisited. In Till Schümmer, editor, *EuroPLoP 2008: 13th Annual European Conference on Pattern Languages of Programming*, volume 610 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[KCM00] Sunwoo Kim, John A. Clark, and John A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *PROC. Net.ObjectDays Conf. Object-Oriented Software Systems*, October 2000.

[KMHSB10] A. Kumar Maji, Kangli Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with android and symbian. In *Proc. of ISSRE'10*, pages 249–258, 2010.

[KP88] G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in smalltalk80 system. *Journal of Object Oriented Programming*, 1:26–49, 1988.

[KVGS11] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. BDTEX: A GQM-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

[KZN12] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[LBB15] Valeria Lelli, Arnaud Blouin, and Baudry Benoit. Classifying and qualifying GUI defects. In *Software Testing, Verification and Validation (ICST), 2015 IEEE Eighth International Conference on*, pages 1–10, April 2015.

[LBB+16] Valeria Lelli, Arnaud Blouin, Baudry Benoit, Fabien Coulon, and Beaudoux Olivier. Automatic detection of GUI design smells: The case of blob listener. In *Submitted to International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[LBBC15] Valeria Lelli, Arnaud Blouin, Baudry Benoit, and Fabien Coulon. On model-based testing advanced GUIs. In *11th Workshop on Advances in Model Based Testing (A-MOST 2015)*, pages 1–10, April 2015.

[Lel13] Valeria Lelli. Challenges of testing for critical interactive systems. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 509–510, March 2013.

[LLS10] Ning Li, Zhanhuai Li, and Xiling Sun. Classification of software defect detected by black-box testing: An empirical study. In *Proc. of WCSE'10*, 2010.

[Lm04] R.R. Lutz and I.C. mikulski. Empirical analysis of safety-critical anomalies during operations. *IEEE Trans. Softw. Eng.*, pages 172–180, 2004.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes –- a comprehensive study on real world concurrency bug characteristics. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, 2008.

[LS07] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7):1120 – 1128, 2007. Dynamic Resource Management in Distributed Real-Time Systems.

[MA07] Kenneth Magel and Izzat Alsmadi. GUI structural metrics and testability testing. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, SEA '07, pages 91–95, Anaheim, CA, USA, 2007. ACTA Press.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[MBN03] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, November 2003.

[MBNR13] Atif Memon, Ishan Banerjee, Bao Nguyen, and Bryan Robbins. The first decade of GUI ripping: Extensions, applications, and broader impacts. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE Press, 2013.

[MC71] E.J. McCluskey and Frederick W. Clegg. Fault equivalence in combinational logic networks. *Computers, IEEE Transactions on*, C-20(11):1286–1293, Nov 1971.

[McC76] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[MEH01] Mark W Maier, David Emery, and Rich Hilliard. Software architecture: introducing ieee standard 1471. *Computer*, 34(4):107–109, 2001.

[Mem03] Atif M. Memon. Advances in GUI testing. In Marvin V. Zelkowitz, editor, *Highly Dependable Software – Advances in Computers*, volume 58, pages 149–201. Academic Press, 2003.

[Mem06] Atif Memon. The GUI Testing FrAmewoRk (GUITAR) project. http://www.cs.umd.edu/ atif/GUITAR-Web/index.html.old, 2006. Online; accessed 26-June-2015.

[Mem07] Atif M Memon. An event-flow model of GUI-based applications for testing. *STVR*, 17(3):137–157, 2007.

[Mem08] Atif M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, November 2008.

[MGDLM10] Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and A Le Meur. DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010.

[MKO02] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, pages 352–, Washington, DC, USA, 2002. IEEE Computer Society.

[MKZD12] Daniel Mauser, Alexander Klaus, Ran Zhang, and Linshu Duan. GUI failure analysis and classification for the development of in-vehicle infotainment. In *Proc. of VALID'12*, pages 79–84, 2012.

[ML09] Mika V. Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, 2009.

[MPN+12] Naouel Moha, Francis Palma, Mathieu Nayrolles, Benjamin Joyen Conseil, Yann-Gael.Gueheneuc@polymtl.Ca Yann-Gael, Guéhéneuc, Benoit Baudry, and Jean-Marc Jézéquel. Specification and Detection of SOA Antipatterns. In *International Conference on Service Oriented Computing*, 2012.

[MPRS11] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. AutoBlackTest: A tool for automatic black-box testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1013–1015. ACM, 2011.

[MPRS12] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. AutoBlackTest: Automatic black-box testing of interactive applications. volume 0, pages 81–90. IEEE Computer Society, 2012.

[Mye91] Brad A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST '91, pages 211–220. ACM, 1991.

[Mye95] Brad A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, mar 1995.

[Nor02] Donald A. Norman. *The Design of Everyday Things*. Basic Books, reprint paperback edition, 2002.

[NPLB09] David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.*, 16(4):1–56, 2009.

[NRBM14] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. GUITAR: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 21(1):65–105, 2014.

[NSS10] Duc Hoai Nguyen, Paul Strooper, and Jörn Guy Süß. Automated functionality testing through GUIs. In *Proceedings of the 33th Australasian Conference on Computer Science*, pages 153–162. Australian Computer Society, 2010.

[OAW+01] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A Fault Model for Subtype Inheritance and Polymorphism. *Software Reliability Engineering, International Symposium on*, 0, 2001.

[OMG07] OMG. UML 2.1.1 Specification, 2007.

[OPM15] Frolin Ocariza, Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, page 11 pages. ACM, 2015.

[OU01] A. Jefferson Offutt and Ronald H. Untch. Mutation testing for the new century. chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[PBDP+14] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *Software Engineering, IEEE Transactions on*, 2014.

[PFV03] Ana C.R. Paiva, João C.P. Faria, and Raul F.A.M. Vidal. Specification-based testing of user interfaces. In *Interactive Systems. Design, Specification, and Verification*, volume 2844 of *Lecture Notes in Computer Science*, pages 139–153. Springer Berlin Heidelberg, 2003.

[PHEG13] Alexander Pretschner, Dominik Holling, Robert Eschbach, and Matthias Gemmar. A generic fault model for quality assurance. In *Proc of MODELS'13*, 2013.

[Pim06] Ana Cristina Ramada Paiva Pimenta. *Automated Specification-Based Testing of Graphical User Interfaces*. PhD thesis, Faculty of Engineering of the University of Porto, November 2006.

[PMP+06] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon v2: Large scale source code analysis and transformation for java. Technical Report hal-01078532, INRIA, 2006.

[Pot96] Mike Potel. MVP: Model-view-presenter the taligent programming model for c++ and java. *Taligent Inc*, 1996.

[Res13a] IBM Research. Extensions for defects in GUI, user documentation, build and national language support (nls). `http://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2-Extensions.pdf`, 2013. Online; accessed 02-June-2015.

[Res13b] IBM Research. Orthogonal defect classification v 5.2 for software design and code. `http://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf`, 2013. Online; accessed 02-June-2015.

[RPV12] David Raneburger, Roman Popp, and Jean Vanderdonckt. An automated layout approach for model-driven wimp-ui generation. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[RRP14] Daniele Romano, Steven Raemaekers, and Martin Pinzger. Refactoring fat interfaces using a genetic algorithm. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 351–360, 2014.

[RT05] F. Ricca and P. Tonella. Web testing: a roadmap for the empirical research. In *Web Site Evolution, 2005. (WSE 2005). Seventh IEEE International Symposium on*, pages 63–70, 2005.

[SCP08] José L. Silva, José Creissac Campos, and Ana C. R. Paiva. Model-based user interface testing with spec explorer and ConcurTaskTrees. *Electron. Notes Theor. Comput. Sci.*, 208:77–93, 2008.

[SCSS14] J.C. Silva, J.C. Campos, J. Saraiva, and J.L. Silva. An approach for graphical user interface external bad smells detection. In Álvaro Rocha, Ana Maria Correia, Felix . B Tan, and Karl . A Stroetmann, editors, *New Perspectives in Information Systems and Technologies, Volume 2*, volume 276 of *Advances in Intelligent Systems and Computing*, pages 199–205. Springer International Publishing, 2014.

[She07] David J. Sheskin. *Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, January 2007.

[Shn83] Ben Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[SKBD14] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-smell detection as a bilevel problem. *ACM Trans. Softw. Eng. Methodol.*, 24(1):6:1–6:44, October 2014.

[SKIH11] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information and Software Technology*, 53(7):774 – 788, 2011.

[SM08] Jaymie Strecker and Atif Memon. Relationships between test suites, faults, and fault detection in GUI testing. In *Proc. of ICST'08*, pages 12–21, 2008.

[Smi09] Josh Smith. WPF apps with the model-view-viewmodel design pattern. *MSDN Magazine*, February 2009.

[SSG+10] João Carlos Silva, Carlos Eduardo Silva, Rui Gonçalo, João Alexandre Saraiva, and José Creissac Campos. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering interactive computing systems (EICS'10)*, pages 181–186, Berlin, Germany, 2010. ACM.

[Sta07] S. Staiger. Static analysis of programs with graphical user interface. In *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, pages 252–264, 2007.

[Sun01] Sun Microsystems. *Java Look and Feel Design Guidelines*. Addison-Wesley, 2 edition, 2001.

[SYA+13] Dag I.K. Sjoberg, Aiko Yamashita, Bente C.D. Anda, Audris Mockus, and Tore Dyba. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[TKH11] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based GUI testing of an android application. In *Software*

*Testing, Verification and Validation (ICST), 2011 IEEE Sixth International Conference on*, ICST '11, pages 377–386. IEEE Computer Society, 2011.

[vBDD+91] Gregor von Bochmann, Anindya Das, Rachida Dssouli, Martin Dubuc, Abderrazak Ghedamsi, and Gang Luo. Fault models in testing. In *Protocol Test Systems*, pages 17–30, 1991.

[vD97] Andries van Dam. Post-WIMP user interfaces. *Commun. ACM*, 40(2):63–67, February 1997.

[vD00] Andries van Dam. Beyond wimp. *IEEE Computer Graphics and Applications*, 20(1):50–51, 2000.

[VGH09] Diego Vallespir, Fernanda Grazioli, and Juliana Herbert. A framework to evaluate defect taxonomies. In *Proceeding of the 15th Argentine Congress on Computer Sciences*, pages 643–652, 2009.

[Wei10] Stephan Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines*. PhD thesis, Humboldt University of Berlin, October 2010.

[Yan11] Xuebing Yang. *Graphic User Interface Modelling and Testing Automation*. PhD thesis, School of Engineering and Science Victoria University, May 2011.

[YCM11] Xun Yuan, M.B. Cohen, and A.M. Memon. GUI interaction testing: Incorporating event context. *Software Engineering, IEEE Transactions on*, 37(4):559–574, 2011.

[YPX13] Wei Yang, MukulR. Prasad, and Tao Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 250–265. Springer Berlin Heidelberg, 2013.

[ZFS15] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103(0):102 – 117, 2015.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

[ZLE12] Sai Zhang, Hao Lü, and Michael D. Ernst. Finding errors in multithreaded GUI applications. In *Proc. ISSTA'12*, ISSTA 2012, pages 243–253. ACM, 2012.

[ZLE13] Sai Zhang, Hao Lü, and Michael D. Ernst. Automatically repairing broken workflows for evolving GUI applications. In *ISSTA 2013, Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 45–55, 2013.

[ZPK14] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *Proc. of ICST'14*, 2014.

# List of Figures

# List of Tables

# List of Algorithms

## Résumé

La communauté du génie logiciel porte depuis ses débuts une attention spéciale à la qualité et la fiabilité des logiciels. De nombreuses techniques de test logiciel ont été développées pour caractériser et détecter des erreurs dans les logiciels. Les modèles de fautes identifient et caractérisent les erreurs pouvant affecter les différentes parties d'un logiciel. D'autre part, les critères de qualité logiciel et leurs mesures permettent d'évaluer la qualité du code logiciel et de détecter en amont du code potentiellement sujet à erreur. Les techniques d'analyses statiques et dynamiques scrutent, respectivement, le logiciel à l'arrêt et à l'exécution pour trouver des erreurs ou réaliser des mesures de qualité.

Dans cette thèse, nous prônons le fait que la même attention doit être portée sur la qualité et la fiabilité des interfaces utilisateurs (ou interface homme-machine, IHM), au sens génie logiciel du terme. Cette thèse propose donc deux contributions dans le domaine du test et de la maintenance d'interfaces utilisateur : 1. Classification et mutation des erreurs d'interfaces utilisateur. 2. Qualité du code des interfaces utilisateur.

Nous proposons tout d'abord un modèle de fautes d'IHM. Ce modèle a été conçu à partir des concepts standards d'IHM pour identifier et classer les fautes d'IHM ; Au travers d'une étude empirique menée sur du code Java existant, nous avons montré l'existence d'une mauvaise pratique récurrente dans le développement du contrôleur d'IHM, objet qui transforme les évènements produits par l'interface utilisateur pour les transformer en actions. Nous caractérisons cette nouvelle mauvaise pratique que nous avons appelée *Blob listener*, en référence à la méthode Blob. Nous proposons également une analyse statique permettant d'identifier automatiquement la présence du *Blob listener* dans le code d'interface Java Swing.



## Abstract

The software engineering community takes special attention to the quality and the reliability of software systems. Software testing techniques have been developed to find errors in code. Software quality criteria and measurement techniques have also been assessed to detect error-prone code.

In this thesis, we argue that the same attention has to be investigated on the quality and reliability of GUIs, from a software engineering point of view. We specifically make two contributions on this topic. First, GUIs can be affected by errors stemming from development mistakes. The first contribution of this thesis is a fault model that identifies and classifies GUI faults. We show that GUI faults are diverse and imply different testing techniques to be detected.

Second, like any code artifact GUI code should be analyzed statically to detect implementation defects and design smells. As for the second contribution, we focus on design smells that can affect GUIs specifically. We identify and characterize a new type of design smell, called Blob listener. It occurs when a GUI listener, that gathers events to treat and transform as commands, can produce more than one command. We propose a systematic static code analysis procedure that searches for Blob listener that we implement in a tool called *InspectorGuidget*. Experiments we conducted exhibits positive results regarding the ability of *InspectorGuidget* in detecting *Blob listeners*. To counteract the use of *Blob listeners*, we propose good coding practices regarding the development of GUI listeners.



N° d'ordre : 15ISAR 26 / D15 - 26



**Institut National des Sciences Appliquées de Rennes**
20, Avenue des Buttes de Coëmes • CS 70839 • F-35708 Rennes Cedex
Tel : 02 23 23 82 00 - Fax : 02 23 23 83 96