HAL
open science

User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners

Arnaud Blouin, Valéria Lelli, Benoit Baudry, Fabien Coulon

To cite this version:

Arnaud Blouin, Valéria Lelli, Benoit Baudry, Fabien Coulon. User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners. Information and Software Technology, 2018, 102, pp.49-64. 10.1016/j.infsof.2018.05.005 . hal-01499106v5

HAL Id: hal-01499106

https://inria.hal.science/hal-01499106v5

Submitted on 29 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

User Interface Design Smell: Automatic Detection and Refactoring of Blob Listeners

Arnaud Blouin[a], Valéria Lelli[b], Benoit Baudry[c], Fabien Coulon[d]

[a] *Univ Rennes, INSA Rennes, Inria, CNRS, IRISA, France*
[b] *Federal University of Ceará, Brazil*
[c] *KTH Royal Institute of Technology, Sweden*
[d] *University of Toulouse - Jean-Jaurès, France*

Abstract

Context. User Interfaces (UIs) intensively rely on event-driven programming: interactive objects send UI events, which capture users' interactions, to dedicated objects called *controllers*. Controllers use several *UI listeners* that handle these events to produce UI commands.
Objective. First, we reveal the presence of design smells in the code that describes and controls UIs. Second, we demonstrate that specific code analyses are necessary to analyze and refactor UI code, because of its coupling with the rest of the code.
Method. We conducted an empirical study on four large Java software systems. We studied to what extent the number of UI commands that a UI listener can produce has an impact on the change- and fault-proneness of the UI listener code. We developed a static code analysis for detecting UI commands in the code.
Results. We identified a new type of design smell, called *Blob listener*, that characterizes UI listeners that can produce more than two UI commands. We proposed a systematic static code analysis procedure that searches for *Blob listener* that we implement in `InspectorGuidget`. We conducted experiments on the four software systems for which we manually identified 53 instances of *Blob listener*. `InspectorGuidget` successfully detected 52 *Blob listeners* out of 53. The results exhibit a precision of 81.25 % and a recall of 98.11 %. We then developed a semi-automatically and behavior-preserving refactoring process to remove *Blob listeners*. 49.06 % of the 53 *Blob listeners* were automatically refactored. Patches have been accepted and merged. Discussions with developers of the four software systems assess the relevance of the *Blob listener*.
Conclusion. This work shows that UI code also suffers from design smells that have to be identified and characterized. We argue that studies have to be conducted to find other UI design smells and tools that analyze UI code must be developed.

Keywords: User interface, Event Handling, Design smell, Software maintenance, Code refactoring, Empirical software engineering

1. Introduction

User Interfaces (UI) are the tangible vector that enable users to interact with software systems. While UI design and qualitative assessment is handled by UI designers, integrating UIs into software systems remains a software engineering task. Software engineers develop UIs following widespread architectural design patterns, such as MVC [1] or MVP [2] (*Model-View-Controller/Presenter*), that consider UIs as first-class concerns (*e.g.,* the *View* in these two patterns). These patterns clarify the implementations of UIs by clearly separating concerns, thus minimizing the "spaghetti of call-backs" [3]. These implementations rely on event-driven programming where events are treated by *controllers* (resp. presenters[1]), as depicted by Listing 1. In this Java Swing code example, the *AController* controller manages three interactive objects, *b1*, *b2*, and *m3* (Lines 2–4). To handle events that these objects trigger in response to users' interactions, the UI listener *ActionListener* is implemented in the controller (Lines 5–16). One major task of UI listeners is the production of UI commands, *i.e.,* a set of statements executed in reaction to a UI event produced by an interactive object (Lines 8, 10, and 14). Like any code artifact, UI controllers must be tested, maintained and are prone to evolution and errors. In particular, software developers are free to develop UI listeners that can produce a single or multiple UI commands. In this work, we investigate the effects of developing UI listeners that can produce one or several UI commands on the code quality of these listeners.

```
1  class AController implements ActionListener {
2    JButton b1;
3    JButton b2;
4    JMenuItem m3;
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      }else if(src instanceof AbstractButton &&
12          ((AbstractButton)src).getActionCommand().equals(
13          m3.getActionCommand()))
14         // Command 3
15      }
16 }}
```

Listing 1: Code example of a UI controller

Email addresses: `ablouin@irisa.fr` (Arnaud Blouin), `valerialelli@great.ufc.br` (Valéria Lelli), `baudry@kth.se` (Benoit Baudry), `fabien.coulon@irit.fr` (Fabien Coulon)

[1] For simplicity, we use the term *controller* to refer to any kind of component of MV* architectures that manages events triggered by UIs, such as *Presenter* (MVP), or *ViewModel* (MVVM [4]).

In many cases UI code is intertwined with the rest of the code. The first step of our work thus consists of a static code analysis procedure that detects the UI commands that a UI listener can produce. Using this code analysis procedure, we then conduct an empirical study on four large Java Swing and SWT open-source UIs: *Eclipse*, *JabRef*, *ArgouML*, and *FreeCol*. We empirically study to what extent the number of UI commands that a UI listener can produce has an impact on the change- or fault-proneness of the UI listener code, considered in the literature as negative impacts of a design smell on the code [5, 6, 7, 8]. The results of this empirical study show evidences that UI listeners that control more than two commands are more error-prone than the other UI listeners. Based on these results, we define a UI design smell we call *Blob listener*, *i.e.,* a UI listener that can produce more than two UI commands. For example with Listing 1, the UI listener implemented in *AController* manages events produced by three interactive objects, *b1*, *b2*, and *m3* (Lines 7, 9, and 12), that produce one UI command each. The empirical and quantitative characterization of the *Blob listener* completes the recent qualitative study on web developers that spots web controllers that do too much as a bad coding practice [9].

Based on the coding practices of the UI listeners that can produce less than three commands, we propose an automatic refactoring process to remove *Blob listeners*.

We provide an open-source tool, `InspectorGuidget`[2], that automatically detect and refactor *Blob listeners* in Java Swing, SWT, and JavaFX UIs. To evaluate the ability of `InspectorGuidget` at detecting *Blob listeners*, we considered the four Java software systems previously mentioned. We manually retrieved all instances of *Blob listener* in each software, to build a ground truth for our experiments: we found 53 *Blob listeners*. `InspectorGuidget` detected 51 *Blob listeners* out of 53. The experiments show that our algorithm has a precision of 88.25 % and recall of 98.11 % to detect *Blob listeners*.

We use the same four software systems to evaluate the ability of `InspectorGuidget` at refactoring *Blob listeners*. `InspectorGuidget` is able to automatically refactor 49.06 % of the 53 *Blob listeners*. We show that two types of *Blob listeners* exist and `InspectorGuidget` is able to refactor 86.7 % of one type of *Blob listeners*. Limitations of the UI toolkits limit the refactoring of the second type of *Blob listeners*. For the four software systems we submitted patches that remove the *Blob listeners* and asked developers for feedback. The patches for *JabRef* and *Freecol* have been accepted and merged. The patches for *Eclipse* are in review. We received no feedback from *ArgoUML* developers. The concept of *Blob listener* and the refactoring solution we propose is accepted by the developers we interviewed.

Our contributions are:

1. an empirical study on four Java Swing and SWT open-source software systems. This study investigates the current coding practices of UI listeners. The main result of this study is the identification of a UI design smell we called *Blob listener*.
2. a precise characterization of the *Blob listener*. We also discuss the different coding practices of UI listeners we observed in listeners having less than three commands.
3. A static code analysis to automatically detect the presence of *Blob listeners* in Java Swing, SWT, and JavaFX UIs.
4. A code refactoring solution to remove *Blob listener* instances.
5. an open-source tool, `InspectorGuidget`, that embeds the code analysis and the code refactoring technique.
6. A quantitative evaluation of the *Blob listener* detection and refactoring techniques.
7. A qualitative evaluation of the *Blob listener* design smell and its refactoring solution. Patches were produced and submitted to the analyzed projects. Projects from which we got answers have accepted and merged the submitted patches. Discussions with concerned developers were conducted on the relevance of: the Blob Listener design smell; the spotted instances of Blob Listener in their code; the refactoring solution.

This paper extends our work published at EICS 2016 [10] with: the refactoring solution, its implementation, and its evaluation; a new algorithm and its implementation for detecting UI command, as the one proposed in [10] contained errors and limitations; a replication of the empirical study using this new implementation and on an improved data set composed of more representative software systems than the ones in [10].

The paper is organized as follows. Section 2 introduces the concept of UI commands and the algorithm for automatically detecting UI commands in UI listeners. Based on the implementation of this algorithm, called `InspectorGuidget`, Section 3 describes an empirical study that investigates coding practices of UI listeners. This study exhibits the existence of an original UI design smell we called *Blob listener*. Section 4 describes the refactoring solution for automatically removing *Blob listeners*. Section 5 evaluates the ability of `InspectorGuidget` in detecting both UI commands and *Blob listeners*, and in refactoring *Blob listeners*. The paper ends with related work (Section 6) and a research agenda (Section 7).

2. Automatic Detection of UI Commands

As any code artifact, UI code has to be validated to find bugs or design smells. Design smells are symptoms of poor software design and implementation choices that affect the quality and the reliability of software systems [11]. If software validation techniques are numerous and broadly used in the industry, they focus on specific programming issues, such as object-oriented issues. We claim that it is necessary to develop specific UI code analysis techniques to take the specific nature of UI code smells and bugs into account. In particular, these analysis techniques must embed rules to extract information specifically about UI code, while this one is deeply intertwined with the rest of the code. These techniques have to extract from the code information and metrics related to UIs intertwined with the rest of the code. In this section, we introduce a code analysis technique for detecting UI commands in the code of Java software systems.

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

2.1. Definitions

In this section we define and illustrate core concepts used in this work.

Definition 1 (UI listener) *UI listeners are objects that follow the event-driven programming paradigm by receiving and treating UI events produced by users while interacting with UIs. In reaction of such events, UI listeners produce UI commands. UI toolkits provide predefined sets of listener interfaces or classes that developers can used to listen specific UI events such as clicks or key pressures.*

This definition follows the concept of *Abstract Listener* from the W3C *Abstract User Interface Models* report that defines a listener as the "*entity used to describe the behavior of [an interactive object] by using Event-Condition-Action (ECA) rules*" [12]. ECA in the context of UI listeners refers to UI events (*Event*), conditional statements that guard the creation of UI commands (*Condition*), and UI commands (*Action*). UI listeners are not limited to graphical user interfaces. UI listeners aim at processing events produced by a user interface, which can be graphical, vocal, command line, *etc.*.
UI commands can be defined as follows:

Definition 2 (UI Command) *A UI command [13, 14], aka.* action *[15, 16], is a set of statements executed in reaction to a user interaction, captured by an input event, performed on a UI. UI commands may be supplemented with pre-conditions checking whether the command fulfills the prerequisites to be executed.*

Programming languages and their different UI toolkits propose various ways to define UI listeners and commands. Figure 1 depicts how UI listeners and commands can be defined in Java UI toolkits. Figure 1 shows the Java Swing code of a controller, *AController*, that manages one button called *b1*. To receive and treat the events produced by users when triggering *b1*, the controller has to register a specific Java Swing listener called *ActionListener* on *b1*. In this example, this listener is the controller itself that implements the *ActionListener* interface. The *actionPerfomed* listener method, declared in *ActionListener*, is called each time the user triggers *b1* to produce a UI command. Before executing the command, verifications may be done to check whether the command can be executed. In our case, the source of the event is compared to *b1* to check that *b1* is at the origin of the event. We call such conditional statements, *source object identification statements*. Then, statements, specified into such conditional statements and that compose the main body of the command, are executed. Statements that may be defined before and after the main body of a command are also considered as part of the command.

UI listeners can be implemented in different ways, which complexifies their code analysis. The three main ones are detailed as follows.

Listeners as anonymous classes – In Listing 2 listeners are defined as an anonymous class (Lines 3–7) and register with one interactive object (Line 2). The methods of this listener are then implemented to define the command to perform when an event occurs. Because such listeners have to handle only one

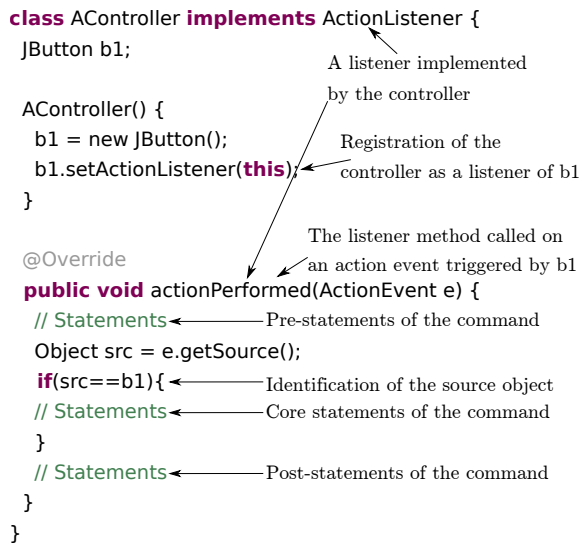

Figure 1: Composition of a UI listener and its UI commands

interactive object, *if* statements used to identify the involved interactive object are not more used, simplifying the code.

Listeners as lambdas – Listing 4 illustrates the same code than Listing 2 but using Lambdas supported since Java 8. Lambdas simplify the implementation of anonymous class that have a single method to implement.

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 3 is a code excerpt that describes a mouse listener where different methods are managed: *mouseClicked* (Line 2), *mouseReleased* (Line 7), and *mouseEntered* (Line 10). Data are shared among these methods (*isDrag*, Lines 3 and 8).

```
1  private void registerInteractiveObjectHandlers() {
2   view.resetPageButton().addActionListener(
3     new ActionListener() {
4     @Override public void actionPerformed(ActionEvent e) {
5       requestData(pageSize, null);
6     }
7   });
8   view.previousPageButton().addActionListener(
9     new ActionListener() {
10      @Override public void actionPerformed(ActionEvent e) {
11        if(hasPreviousBookmark())
12          requestData(pageSize, getPreviousBookmark());
13      }
14    });//...
15 }
```

Listing 2: UI listeners as anonymous classes

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

Listing 3: A UI listener defined as a class

```
private void registerInteractiveObjectHandlers() {
 view.resetPageButton().addActionListener(
              e -> requestData(pageSize, null));

 view.previousPageButton().addActionListener(e -> {
  if (hasPreviousBookmark())
    requestData(pageSize, getPreviousBookmark());
 });

 //...
}
```

Listing 4: Same code than in Listing 2 but using Java 8 Lambdas

2.2. Detecting UI commands

This section details the algorithm for statically detecting UI commands in source code. Algorithm 1 summarizes the UI command detection process. The detection process includes three main steps. First, UI listeners are identified in the code. This trivial step consists of collecting the classes that implements UI listener interfaces provided by the supported UI toolkits (Line 2). Second, from the body statements of each detected listener class, *source object identification statements* are identified (*cf.* Section 2.2.1). Third, the statements that compose the different commands managed in one listener are identified (*cf.* Section 2.2.2).

Algorithm 1 UI command detection

Input: *classes*, the source classes of the software system
Input: *tkListeners*, the listener classes/interfaces of supported UI toolkits
Output: *cmds*, the detected UI commands

1: $cmds \leftarrow \varnothing$
2: $uiListeners \leftarrow findUIListeners(classes, tkListeners)$
3: **foreach** $l \in uiListeners$ **do**
4: $s \leftarrow l.getBody()$
5: $objectIdentifs \leftarrow findSrcObjectIdentificationStatmts(s)$
6: **if** $objectIdentifs == \varnothing$ **then**
7: $cmds \leftarrow cmds \cup \{Command(s, \varnothing\}$
8: **else**
9: **foreach** $i \in objectIdentifs$ **do**
10: $cmds \leftarrow$
11: $cmds \cup \{extractCmdFromObjectIdentif(i, s)\}$
12:
13: **function** FINDSRCOBJECTIDENTIFICATIONSTATMTS($statmts$)
14: $identifs \leftarrow findConditionalStatmts(statmts)$
15: $identifs \leftarrow filterConditionalsThatUseUIEvent(identifs)$
16: $identifs \leftarrow filterLastNestedConditionals(identifs)$
17: **return** $identifs$
18:
19: **function** EXTRACTCMDFROMOBJECTIDENTIF($conds$, $statmts$)
20: $s \leftarrow conds.getStatmts()$
21: $s \leftarrow s \cup backwardSlicing(conds.getStatmts())$
22: $s \leftarrow s \cup forwardSlicing(conds.getStatmts())$
23: $considerDispatchedMethods(s)$
24: **return** $Command(s, conds)$

2.2.1. Identifying source object identification statements

Software developers are free to develop UI listeners that can produce a single or multiple UI commands. To identify the interactive object at the origin of a given event, conditional statements are commonly used. We call such conditional statements, *source object identification statements*. Such conditional blocks may encapsulate a command to execute in reaction to the event. Their identification is therefore mandatory to detect UI commands. For example, five nested conditional blocks (Lines 7, 9, 10, 12, and 14) compose the listener method *actionPerformed* in Listing 5. The first conditional block checks the type of the interactive object that produced the event (Line 7):

```
if(src instanceof JMenuItem || src instanceof JButton)
```

This block contains three other conditional blocks that identify the interactive object using its action command (Lines 9, 12, and 14), for example:

```
if(cmd.equals("Copy"))
```

Each of these three blocks encapsulates one command to execute in reaction of the event.

As summarized in Algorithm 1 (Lines 13–17), the detection of source object identification statements starts by identifying all the conditional statements contained in the given listener method. Then, only the conditional statements that directly or indirectly use the UI event given as a parameter of the listener method are considered. For example `if(selectedText)` (Line 10) makes no use of the UI event *e* and is thus not considered as a source object identification statement. Finally, on nested conditional statements, the last nested one is considered as the main source object identification of a command.

We empirically identified three kinds of source object identification statements by looking an existing Java code, as explained as follows.

Comparing a property of the interactive object – Listing 5 is an example of the first variant of source object identification statements: the interactive object that produced the event (lines 9, 12, and 14) are identified with a string value associated to the interactive object and returned by *getActionCommand* (line 8). Each of the three *if* blocks forms a UI *command* to execute in response of the event produced by interacting with a specific interactive object (lines 10, 11, 13, and 15).

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

Listing 5: interactive object identification using interactive object's properties in Swing

In Java Swing, the properties used to identify interactive objects are mainly the *name* or the *action command* of these interactive objects. The action command is a string value used to identify the source interactive object using a UI event. Listing 6, related to Listing 5, shows how an action command (lines 2 and 6) and a listener (lines 3 and 7) can be associated to an interactive object 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 6: Initialization of Swing interactive objects to be controlled by the same listener

Checking the type of the interactive object – The second variant of source object identification statements consists of checking the *type* of the interactive object that produced the event. Listing 7 depicts such a practice where the type of the interactive object 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 interactive object 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 }}
```

Listing 7: interactive object identification using the operator *instanceof*

Comparing interactive object references – The last variant consists of comparing interactive object references to identify those at the origin of the event. Listing 8 illustrates this variant where *getSource* returns the source interactive object of the event that is compared to interactive object 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    }}
```

Listing 8: Comparing interactive object references

In these three variants, multiple *if* statements are successively defined. Such successions are required when one single UI listener gathers events produced by several interactive objects. In this case, the listener needs to identify the interactive object that produced the event to process. When no source object identification statement is detected in a UI listener, all the statements of the listener are considered as part of a unique command (Lines 6 and 7 in Algorithm 1).

The three variants of source object identification statements appear in all the main Java UI toolkits, namely Swing, SWT, GWT, and JavaFX. Examples for these toolkits are available on the companion webpage of this paper[2].

2.2.2. *Extracting UI command statements*

We consider that each source object identification statement surrounds a UI command. From a source object identification statement, the code statements that compose the underlying UI command are identified as follows (Lines 19–24, Algorithm 1). First, the statements that compose the conditional statement spotted as a source object identification statement are collected and considered as the main statements of the command. Second, these statements may depend on variables or attributes previously defined and used. A static backward slicing [17] is done to gather all these code elements and integrate them into the command. For example, the following code illustrates the statements sliced for the first of the two commands of the listener. The statements lines 2 and 4 are sliced since they are used by the source object identification statements of command #1: *s.equals("Copy")*

```
1  public void actionPerformed(ActionEvent evt) {
2   Object src = evt.getSource(); // sliced
3    if(src instanceof JMenuItem){ // Object identification
4     String s = evt.getActionCommand(); // sliced
5    if(s.equals("Copy")){ // Cmd #1 Object identification
6     if(selectedText) // Main command statement
7      output.copy(); // Main command statement
8    }else if(s.equals("Cut")){ // Cmd #2 Object identification
9     output.cut(); // Not considered in cmd #1
10   }
11   output.done(); // sliced
12   }
13 }
```

Similarly, a static forward slicing is done to gather all the code elements related to the command defined after the source object conditional statement. For example, the statement located Line 11 is sliced since *output* is used by Command #1. The statements part of the main statements of another command are not sliced (*e.g.,* Line 9).

Some listeners do not treat the UI event but dispatch this process to another method, called *dispatch method*. The following code excerpt depicts such a case. Each method invocation that takes as argument the event or an attribute of the event is considered as a dispatch method (*e.g.,* Lines 2 and 6). In this case, the content of the method is analyzed similarly to the content of a listener method.

```
1 public void actionPerformed(ActionEvent evt) {
2  treatEvent(evt);
3 }
4
5 private void treatEvent(AWTEvent evt) {
6  Object src = evt.getSource();
7  //...
8 }
```

Table 1: The four selected software systems and their characteristics

Software system	Version	UI toolkit	kLoCs	# commits	# UI listeners	Source repository link
Eclipse (platform.ui.workbench)	4.7	SWT	143	10049	259	https://git.eclipse.org/c/gerrit/platform/eclipse.platform.git/
JabRef	3.8.0	Swing	95	8567	486	https://github.com/JabRef/jabref
ArgoUML	0.35.1	Swing	101	10098	214	https://github.com/rastaman/argouml-maven http://argouml.tigris.org/source/browse/argouml/
FreeCol	0.11.6	Swing	118	12330	223	https://sourceforge.net/p/freecol/git/ci/master/tree/

3. An empirical study on UI listeners

UI listeners are used to bind UIs to their underlying software system. The goal of this study is to *state whether the number of UI commands that UI listeners can produce has an effect on the code quality of these listeners*. Indeed, software developers are free to develop UI listeners that can produce a single or multiple UI commands since no coding practices or UI toolkits enforce coding recommendations. To do so, we study to what extent the number of UI commands that a UI listener can produce has an impact on the change- and fault-proneness of the UI listener code. Such a correlation has been already studied to evaluate the impact of several antipatterns on the code quality [6]. Change- and fault-proneness are considered in the literature as negative impacts of a design smell on the code [5, 6, 7, 8, 18].

We formulate the research questions of this study as follows:

RQ1 To what extent the number of UI commands per UI listeners has an impact on fault-proneness of the UI listener code?

RQ2 To what extent the number of UI commands per UI listeners has an impact on change-proneness of the UI listener code?

RQ3 Do developers agree that a threshold value, *i.e.,* a specific number of UI commands per UI listener, that can characterize a UI design smell exist?

To answer these three research questions, we measured the following independent and dependent variables. All the material of the experiments is freely available on the companion web page[2].

3.1. Tool

The command detection algorithm has been implemented in `InspectorGuidget`, an open-source Eclipse plug-in that analyzes Java Swing, SWT, and JavaFX software systems[2]. `InspectorGuidget` uses *Spoon*, a library for transforming and analyzing Java source code [19], to support the static analyses.

3.2. Independent Variables

Number of UI Commands (**CMD**). This variable measures the number of UI commands a UI listener can produce. To measure this variable, we use the proposed static code analysis algorithm detailed in Section 2 and implemented in `InspectorGuidget`.

3.3. Dependent Variables

Average Commits (**COMMIT**). This variable measures the average number of commits of UI listeners. This variable will permit to evaluate the change-proneness of UI listeners. To measure this variable, we automatically count the number of the commits that concern each UI listener.

Average fault Fixes (**FIX**). This variable measures the average number of fault fixes of UI listeners. This variable will permit to evaluate the fault-proneness of UI listeners. To measure this variable, we manually analyze the log of the commits that concern each UI listener. We manually count the commits which log refers to a fault fix, *i.e.,* logs that point to a bug report of an issue-tracking system (using a bug ID or a URL) or that contain the term "*fix*" (or a synonymous). We use the following list of terms to identify a first list of commits: *fix, bug, error, problem, work, issue, ticket, close, reopen, exception, crash, NPE, IAE, correct, patch, repair, rip, maintain, warning*. We then manually scrutinized each of these commits. For example, the following commit message extracted from the Eclipse history is considered as a bug fix as it fixes a Java exception (*NullPointerException*, *NPE*): "*49216 [About] NPE at ...AboutFeaturesDialog.buttonPressed*". However, we do not consider the following commit "*Bug 509477 - Use lambdas in ...ui.workbench"* as a bug fix. The term "bug" is used here as a reference to the issue tracking system and not to a real error.

Both COMMIT and FIX rely on the ability to get the commits that concern a given UI listener. For each software system, we use all the commits of their history as the time-frame of the analysis. We ignore the first commit as it corresponds to the creation of the project.

The size, *i.e.,* the number of lines of code (LoC), of UI listeners may have an impact on the number of commits and fault fixes. So, we need to compare UI listeners that have a similar size by computing the four quartiles of the size distribution of the UI listeners [20, 9]. We kept the fourth quartile (Q_4) as the single quartile that contains enough listeners with different numbers of commands to conduct the study. This quartile Q_4 contains 297 UI listeners that have more than 10 lines of code. For the study the code has been formatted and the blank lines and comments have been removed.

Commits may change the position of UI listeners in the code (by adding or removing LoCs). To get the exact position of a UI listener while studying its change history, we use the Git tool

git-log[3]. The *git-log* tool has options that permit to: follow the file to log across file renames (option *-M*); trace the evolution of a given line range across commits (option *-L*). We then manually check the logs for errors.

3.4. Objects

The objects of this study are open-source software systems. The dependent variables, previously introduced, impose several constraints on the selection of these software systems. They must use an issue-tracking system and the Git version control system. We focused on software systems that have more than 5000 commits in their change history to let the analysis of the commits relevant. In this work, we focused on Java Swing and SWT UIs because of the popularity and the large quantity of Java Swing and SWT legacy code available on code repositories such as *Github*[4] and *Sourceforge*[5]. We thus selected four Java Swing and SWT software systems, namely ArgoUML, JabRef, Eclipse (more precisely the *platform.ui.workbench* plug-in), and Freecol. Table 1 lists these software systems, the version used, their UI toolkit, their number of Java line of codes, commits, and UI listeners, and the link the their source code. The number of UI listeners excludes empty listeners. The average number of commits of these software systems is approximately 10.2k commits. The total size of Java code is 457k Java LoCs, excluding comments and blank lines. Their average size is approximately 114k Java LoCs.

3.5. Results

We can first highlight that the total number of UI listeners producing at least one UI command identified by our tool is 1205, *i.e.,* an average of 301 UI listeners per software system. This approximately corresponds to 11 kLoCs of their Java code.

As explained in Section 3.3, to compare listeners with similar sizes we used the quartile Q_4 for the study. Figure 2 shows the distribution of the listeners of Q_4 according to their number of UI commands. 69.36 % (*i.e.,* 206) of the listeners can produce one command (we will call them one-command listeners). 30.64 % of the listeners can be produce two or more commands: 47 listeners can produce two commands. 16 listeners can produce three commands. 28 listeners can produce at least four commands. To obtain representative data results, we considered in the following analyses three categories of listeners: *one-command listener* (*CMD=1* in Table 2), *two-command listener* (*CMD=2*), *three+-command listener* (*CMD>=3*).

We computed the means of *FIX* and *COMMIT* for each of these three categories. To compare the effect size of the means (*i.e., CMD=1 vs. CMD=2*, *CMD=1 vs CMD=2*, *and CMD=1 vs. CMD>=3*) we used the Cohen's *d* index [21]. Because we compared multiple means, we used the Bonferroni-Dunn test [21] to adapt the confidence level we initially defined at 95 % (*i.e.,* $\alpha = 0.05$): we divided this α level by the number of comparisons (3), leading to $\alpha = 0.017$. We used the following code scheme to report the significance of the computed *p*-value: No significance= $p > 0.017$, $* = p \leq .0017$, **= $p \leq .005$, $*** = p \leq .001$. Because *FIX* (resp. *COMMIT*) and *CMD* follow a linear relationship, we used the Pearson's correlation coefficient to assess the correlation between the number of fault fixes (resp. number of changes) and the number of UI commands in UI listeners [21]. The correlation is computed on all the data of Figure 2 (*i.e.,* not using the three categories of listeners). The results of the analysis are detailed in Table 2.

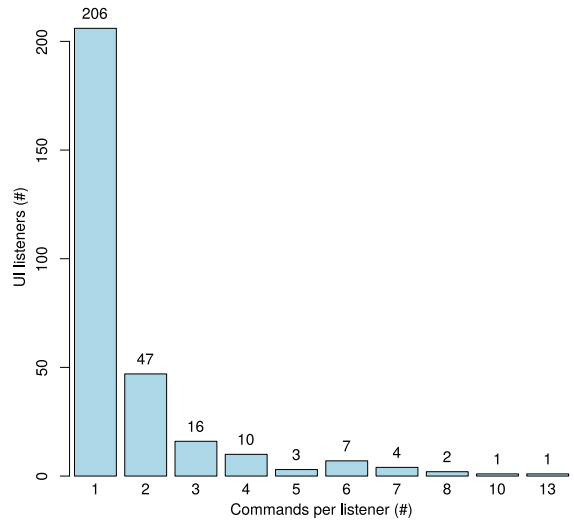

Figure 2: Distribution of the listeners according to their number of UI commands

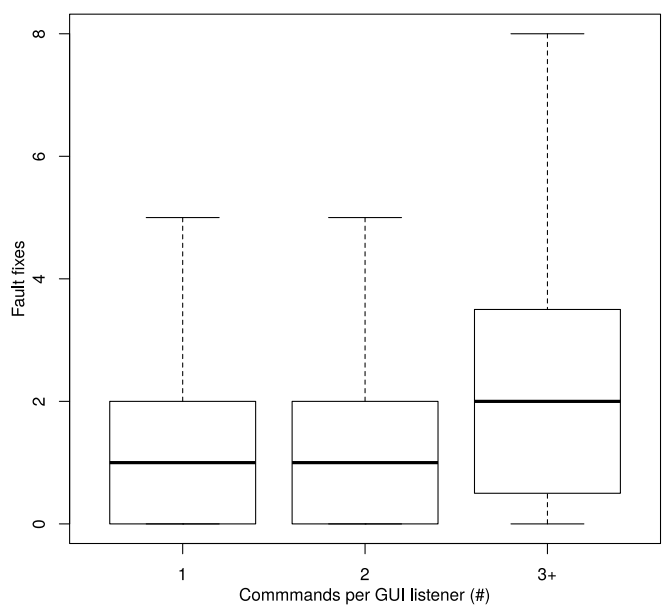

Figure 3: Number of fault fixes of UI listeners

Figure 3 depicts the evolution of *FIX* over *CMD*. We observe a significant increase of the fault fixes when $CMD \geq 3$. According to the Cohen's *d* test, this increase is large (0.8148). *FIX* increases over *CMD* with a moderate correlation (0.4281, if in $[0.3, 0.7[$, a correlation is considered to be moderate [21]).

[3] https://git-scm.com/docs/git-log
[4] https://github.com/
[5] https://sourceforge.net/

Table 2: Means, correlations, and Cohen's *d* of the results

Dependent variables	Mean CMD=1	Mean CMD=2	Mean CMD>=3	Correlation (significance)	Cohen's d CMD=1 vs CMD=2 (significance)	Cohen's d CMD=2 vs CMD>=3 (significance)	Cohen's d CMD=1 vs CMD>=3 (significance)
FIX	1.107	1.149	2.864	0.4281 (***)	0.0301 (no)	0.5751 (no)	0.8148 (***)
COMMIT	5.854	6.872	10.273	0.3491 (***)	0.1746 (no)	0.3096 (no)	0.5323 (no)

Regarding **RQ1**, on the basis of these results we can conclude that managing several UI commands per UI listener has a negative impact on the fault-proneness of the UI listener code: a significant increase appears at three commands per listener, compared to one-command listeners. There is a moderate correlation between the number of commands per UI listener and the fault-proneness.

Figure 4 depicts the evolution of *COMMIT* over *CMD*. The mean value of *COMMIT* increases over *CMD* with a weak correlation (0.3491, using the Pearson's correlation coefficient). A medium (Cohen's *d* of 0.5323) but not significant (*p*-value of 0.0564) increase of COMMIT can be observed between one-command and three+-command listeners. *COMMIT* increases over *CMD* with a moderate correlation (0.3491).

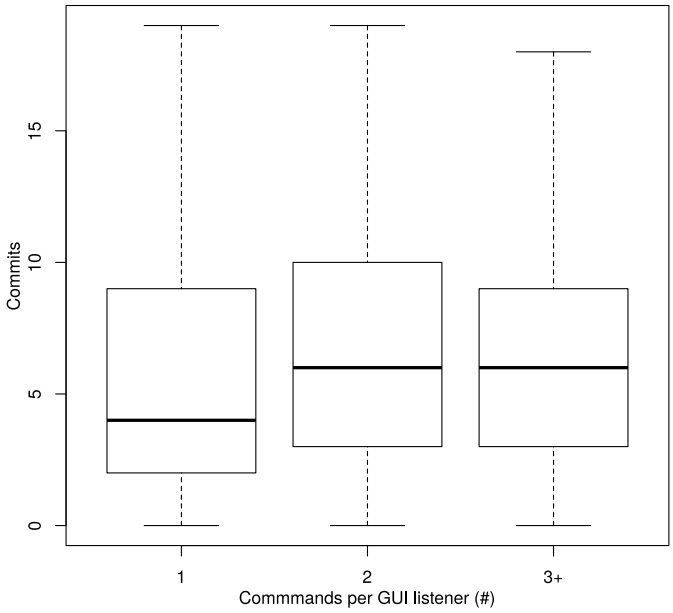

Figure 4: Number of commits of UI listeners

Regarding **RQ2**, on the basis of these results we can conclude that managing several UI commands per UI listener has a small but not significant negative impact on the change-proneness of the UI listener code. There is a moderate correlation between the number of commands per UI listener and the change-proneness.

Regarding **RQ3**, we observe a significant increase of the fault fixes for three+-command listeners against one-command listeners. We also observe an increase of the commits for three+-command listeners against one-command listeners. We thus state that a threshold value, *i.e.,* a specific number of UI commands per UI listener, that characterizes a UI design smell exists. Note that since the COMMIT metrics counts all the commits, bug-fix commits included, the increase of the commits may be correlated to the increase of the fault fixes for three+-command listeners. We contacted developers of the analyzed software systems to get feedback about a threshold value. Beyond the "*sounds good*" for three commands per listener, one developer explained that "*strictly speaking I would say, more than one or two are definitely an indicator. However, setting the threshold to low* [lower than three commands per listener] *could lead to many false positives*". Another developer said "*more than one [command per listener] could be used as threshold, but generalizing this is probably not possible*". We agree and *define the threshold to three UI commands per UI listener*. Of course, this threshold value is an indication and as any design smell it may vary depending on the context. Indeed, as noticed in several studies, threshold values of design smells must be customizable to let system experts the possibility to adjust them [22, 23].

The threats to validity of this empirical study are discussed in Section 5.4.

3.6. *Introducing the Blob Listener design smell*

Based on the results of the empirical study previously detailed, we showed that a significant increase of the fault fixes and changes for two- and three+-command listeners is observed. Considering the feedback from developers of the analyzed software systems, we define at three commands per listener the threshold value from which a design smell, we called *Blob listener*, appears. We define the *Blob listener* as follows:

Definition 3 (Blob Listener) *A* Blob listener *is a UI listener that can produce several UI commands.* Blob listeners *can produce several commands because of the multiple interactive objects 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) identify the interactive object that produced the UI event to treat; 2) execute the corresponding UI command.*

4. Refactoring *Blob listeners*

This section details the semi-automatic and behavior preserving code refactoring technique [24, 11] for removing *Blob*

```
class B implements ActionListener {
  JButton but1;
  JButton but2;

  B() {
      but1 = new JButton( text: "button1");
      but1.setActionCommand("BUTTON1_ACTION_CMD");
      but1.addActionListener( l: this);

      but2 = new JButton( text: "button2");
      but2.setActionCommand("BUTTON2_ACTION_CMD");
      but2.addActionListener( l: this);
  }

  @Override
  public void actionPerformed(final ActionEvent e) {
      if(e.getActionCommand().equals( anObject: "BUTTON1_ACTION_CMD")) {
          System.out.println( x: "Command 1");
          return;
      }
      if(e.getActionCommand().equals( anObject: "BUTTON2_ACTION_CMD")) {
          System.out.println( x: "Command 2");
          return;
      }
  }
}
```

```
7   class B {
8       JButton but1;
9       JButton but2;
10
11      B() {
12          but1 = new JButton( text: "button1");
13          but1.addActionListener(e -> System.out.println( x: "Command 1"));
14          but2 = new JButton( text: "button2");
15          but2.addActionListener(e -> System.out.println( x: "Command2 "));
16      }
17  }
18
```

Figure 5: Simple example of the refactoring effects. On the left, the original source code. On the right the refactored source code

listeners from source code. Algorithm 2 details the refactoring process. The general idea of the refactoring is to move each command that compose a *Blob listener* into a new UI listener directly applied on the targeted interactive object. Figure 5 illustrates the refactoring using a mere example. The original UI listener of this example (Lines 22-30, on the left) manages two commands that can be produced by one interactive object each, namely *but1* and *but2*. These two interactive objects register the listener Lines 14 and 18. String values are used to identify the interactive object at the origin of the UI event (Lines 13, 17, 23, and 27). Each command that composes a *Blob listener* (identified using the static code analysis detailed in Section 2) are moved into a new UI listener directly at the listener registration of the interactive object (Lines 13 and 15, on the right), as detailed in the following algorithm.

Algorithm 2 *Blob listener* refactoring

Input: *classes*, the source classes of the software system
Input: *blobs*, The *Blob listener* spotted in the source code

```
allObjs ← findAllInteractiveObjects(classes)
foreach blob ∈ blobs do
    foreach cmd ∈ blob.getCommands() do
        interObjects ← findAssociatedInterObjects(cmd, allObjs)
        foreach obj ∈ interObjects do
            registration ← findListenerRegistration(obj)
            removeUselessStatements(cmd)
            newListener ← new UI Listener
            replaceListenerRegistration(registration, newListener)
            copyStatements(cmd.getStatmts(), newListener)
            removeOldCommand(cmd, obj)
```

The first step of the algorithm identifies all the interactive objects declared in the source code and their usages (Section 4.1). Then, the identification of the interactive objects associated to (*i.e.,* that can produce) each command of a *Blob listener* is done (Section 4.2). Following, for each associated interactive object found, a new UI listener object is created using the command statements. This new listener is then directly used at the listener registration of the interactive object (Section 4.3). Finally, the *Blob listener* is removed (Section 4.4).

4.1. Finding all the interactive objects and their usages

As illustrated with Listings 5 and 6, the definition of a *Blob listener* (*e.g.,* Listing 5) and its registrations to different interactive objects (*e.g.,* Listing 6) are separated in the code. Refactoring *Blob listeners* first requires the identification of the interactive objects that register to the *Blob listeners*. To do so, a static code analysis scrutinizes the code to identify local variables and class attributes that are instances of UI toolkits interactive objects. Then, the initialization statements of each interactive object, that we call *interactive object usages*, are identified.

```
menuItem = new JMenuItem();
menuItem.setActionCommand("COPY");
menuItem.addActionListener(listener);
```

For example the three statements of the code above are considered as interactive objects usages of *menuItem* since they initialize these statements. In the method that initializes a given interactive object, we consider all the statements using this interactive object as initialization statements. The last initialization statement of these three lines of code is the registration of a UI listener (*addActionListener*) to *menuItem*. Such a statement permits the identification of the interactive objects involved in the different UI listeners. The next sub-section details how for each command of a given UI listener, interactive objects can be precisely identified.

4.2. Finding associated interactive objects

Section 2 details how the statements and the source object identification statements that compose a UI command are identified. Based on the source object identification statements of each interactive object, we developed a static code analysis for identifying the potential interactive objects that may trigger the command among all the interactive objects of the software system. As detailed in Section 2.2.1, we identified three kinds of source object identification statements. The proposed analysis considers these three kinds to precisely identify the underlying interactive objects plus another technique, as explained as follows.

Comparing interactive object references – Interactive object variables are compared to the object at the origin of the UI event, for example:

```
if(event.getSource() == view.moveDown)
```

In this simple case, we collect all the interactive objects used in the source object identification statements.

Comparing a property of the interactive object – The identification of the source interactive object may use some of its properties, such as its name or its action command. For example a string literal can be used to identify an interactive object:

```
if(event.getActionCommand().equals("COPY"))
```

The analysis searches for string literals used both in the source object identification statements of a command and the usages of all the interactive objects. The following code excerpt illustrates this step where *button* uses the string value *"COPY"*:

```
button.setActionCommand("COPY");
```

The use of string constants is also supported, for example:

```
public final String ACTION_CMD_COPY = "COPY";
//...
if(event.getActionCommand().equals(ACTION_CMD_COPY)) //...
```

Checking the type of the interactive object – Checking the type of the source object that produced the event can help in detecting the underlying interactive objects. The following source object identification statement implies that the underlying interactive object is a *JButton*.

```
if(event.getSource() instanceof JButton)
```

This technique helps in reducing the number of candidates but may not precisely identify the underlying interactive object if several interactive objects of the same type are managed by the same *Blob listener*.

Analyzing listener registrations – As explained in Section 2, UI listener can be directly implemented during their registration to an interactive object. The following code excerpt illustrates this case where the listener is implemented as an anonymous class.

```
view.resetPageButton().addActionListener(
   new ActionListener() {
   @Override public void actionPerformed(ActionEvent e) {
     requestData(pageSize, null);
   }
});
```

In this case, the interactive object can be directly identified by looking at the registration method invocation. However, since this practice permits the registration of a listener to a unique interactive object, it does not fit our case since *Blob listeners* manage several interactive objects.

Once the involved interactive objects identified for each command, they are checked against the interactive objects that register the UI listener (Section 4.1). This step aims at checking that the interactive objects used by a command and the interactive objects that register the UI listener of the command are the same. Then, the refactoring is applied as detailed in the next section. If no interactive object is identified neither in the listener registrations nor in the command statements, a warning is raised and the refactoring stopped.

4.3. Fragmenting a Blob listener *in atomic UI listeners*

As illustrated by Figure 5, the goal of the refactoring process is to fragment a *Blob listener* into several atomic UI listeners directly defined at the listener registration (one atomic UI listener for each associated interactive object). To do so, the developer can choose to either refactor *Blob listeners* as Lambdas (concise and lightweight but requires Java 8) or as anonymous classes (more verbose but supported by all the Java versions). As summarized in Algorithm 2, a new UI listener is created for each interactive object of each UI command of a given *Blob listener* (Line 8, Algorithm 2). Each new atomic UI listener registers their targeted interactive object (Line 9, Algorithm 2) as follows:

```
interactiveObject.addActionListener(evt -> {
   // Command statements
});
```

Then, the statements that form the UI command are copied into this new atomic UI listener (Line 10, Algorithm 2):

```
interactiveObject.addActionListener(evt -> {
   output.copy();
});
```

All the source object identification statements (*e.g.,* Lines 23 and 27 in Figure 5) are removed from the command statements since they are no more necessary. Statements, such as *return* statements (*e.g.,* Lines 25 and 29 in Figure 5) that may end a UI command are also removed. In some cases, the refactoring cannot be automatic and requires decisions from the developer, as discussed in Section 4.5.

In some cases, several interactive objects can be associated to a single UI command. For example, the following code excerpt shows that two different interactive objects *button* and *menu* (Lines 8–9) perform the same command (Line 13).

```
1  class Controller implements ActionListener {
2     static final String ACTION_CMD = "actionCmd";
3     JButton button;
4     JMenuItem menu;
5
6     Controller() {
7        //...
8        button.setActionCommand(ACTION_CMD);
9        menu.setActionCommand(ACTION_CMD);
10    }
11
12    public void actionPerformed(ActionEvent e) {
13       if(ACTION_CMD.equals(e.getActionCommand())) {
14          // A single command for 'button' and 'menu'
15       }
16    }
17 }
```

In such a case, the refactoring process creates one constant attribute that contains the shared UI listener (Line 5 in the following code excerpt). Each interactive object registers the same listener (Lines 8 and 9). This techniques does not contradict the *Blob listener* design smell (*i.e.,* UI listeners that can produce several commands) since the same command is produced by several interactive objects.

```
class Controller {
   JButton button;
   JMenuItem menu;

   final ActionListener listener = e -> {/* The command */};

   Controller() {
      button.setActionListener(listener);
      menu.setActionListener(listener);
   }
}
```

4.4. *Removing a* Blob listener

Once a *Blob listener* fragmented into several atomic UI listeners, the *Blob listener* is removed from the source code. As illustrated by Figure 5, the UI listener method that forms the *Blob listener* is removed (here, *actionPerformed*). The class that contained the *Blob listener* does not implement the UI listener interface anymore (here, *ActionListener*). Statements of the initialization of the involved interactive objects may be removed if no more used and if used to identify the source interactive object in the *Blob listener*. For Java Swing, such statements are typically invocations of the *setActionCommand* method.

4.5. *Limits of the refactoring process*

In some specific cases the refactoring cannot be done automatically. For example, the following code excerpt shows a UI command (Line 12) that makes use of a class attribute (Line 3) initialized Lines 6–7. This listener does not define and register the interactive objects. This job is done in another class.

```
1  class Controller implements ActionListener {
2    static final String ACTION_CMD = "actionCmd";
3    JFileChooser c;
4
5    public Controller() {
6      c = new JFileChooser();
7      c.setMultiSelectionEnabled(false);
8    }
9
10    public void actionPerformed(ActionEvent e) {
11       if(ACTION_CMD.equals(e.getActionCommand())) {
12          c.showDialog(null, "title");
13       }
14    }
15 }
```

By default the class attributes (here *c* and its initialization statements) used in UI commands are copied in the class that contains the interactive objects, as shown in the following refactored code that made use of the previous listener:

```
class A {
  JFileChooser c;
   JButton b;

  public A() {
    c = new JFileChooser();
    c.setMultiSelectionEnabled(false);
    b = new JButton();
    b.setActionListener(e -> c.showDialog(null, "title"));
  }
}
```

This strategy may not be relevant in certain situations and the developer has to validate this change or finalize the refactoring. We did not face this situation during our experiments of Section 5.

5. Evaluation

To evaluate the efficiency of our detection algorithm and refactoring process, we addressed the four following research questions:

RQ4 To what extent is the detection algorithm able to detect UI commands in UI listeners correctly?

RQ5 To what extent is the detection algorithm able to detect *Blob listeners* correctly?

RQ6 To what extent does the refactoring process propose correct refactoring solutions?

RQ7 To what extent the concept of *Blob listener* and the refactoring solution we propose are relevant?

The evaluation has been conducted using `InspectorGuidget`, our implementation of the *Blob listener* detection and refactoring algorithms introduced in Section 3.1. `InspectorGuidget` leverages the Eclipse development environment to raise warnings in the Eclipse Java editor on detected Blob listeners and their UI commands. The refactoring process is performed outside the Eclipse environment. Initial tests have been conducted on software systems not reused in this evaluation. `InspectorGuidget` allows the setting of this threshold value to let system experts the possibility to adjust them, as suggested by several studies [22, 23]. For the evaluation, the threshold has been set to two-commands per listener. `InspectorGuidget` and all the material of the evaluation are freely available on the companion web page[2].

5.1. *Objects*

We conducted our evaluation using the four large open-source software systems detailed in Section 3.

5.2. *Methodology*

The accuracy of the static analyses that compose the detection algorithm is measured by the *recall* and *precision* metrics [18]. We ran `InspectorGuidget` on each software system to detect UI listeners, commands, and *Blob listeners*. We assumed as a precondition that only UI listeners are correctly identified by our tool. Thus, to measure the precision and recall of our automated approach, we manually analyzed all the UI listeners detected by `InspectorGuidget` to:

Check commands. We manually analyzed each UI listeners to state whether the UI commands they contain are correctly identified. The *recall* measures the percentage of correct UI commands that are detected (Equation (1)). The *precision* measures the percentage of detected UI commands that are correct (Equation (2)). For 39 listeners, we were not able to identify the commands of UI listeners. We removed these listeners from the data set.

$$recall_{cmd}(\%) = \frac{|\{correctCmds\} \cap \{detectedCmds\}|}{|\{correctCmds\}|} \times 100 \qquad (1)$$

$$precision_{cmd}(\%) = \frac{|\{correctCmds\} \cap \{detectedCmds\}|}{|\{detectedCmds\}|} \times 100 \qquad (2)$$

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

Check Blob Listeners. This analysis directly stems from the UI command one since we manually checked whether the detected *Blob listeners* are correct with the threshold value of three commands per UI listener. We used the same metrics used for the UI command detection to measure the accuracy of the *Blob listeners* detection:

$$recall_{blob}(\%) = \frac{|\{correctBlobs\} \cap \{detectedBlobs\}|}{|\{correctBlobs\}|} \times 100 \quad (3)$$

$$precision_{blob}(\%) = \frac{|\{correctBlobs\} \cap \{detectedBlobs\}|}{|\{detectedBlobs\}|} \times 100 \quad (4)$$

5.3. Results and Analysis

RQ4: Command Detection Accuracy. Table 3 shows the number of commands successfully detected per software system. `InspectorGuidget` detected 1392 of the 1400 UI commands (eight false negatives), leading to a recall of 99.43 %. `InspectorGuidget` also detected 62 irrelevant commands, leading to a precision of 95.73 %.

Table 3: Command detection results

Software System	Successfully Detected Commands (#)	FN (#)	FP (#)	$Recall_{cmd}$ (%)	$Precision_{cmd}$ (%)
Eclipse	330	0	5	100	98.51
JabRef	510	5	7	99.03	98.65
ArgoUML	264	3	3	98.88	98.88
FreeCol	288	0	47	100	85.93
OVERALL	1392	8	62	99.43	95.73

Figure 6 classifies the 70 FN and FP commands according to their underlying issues. 44 FP commands are classified as *command parameters*. In these cases, several commands spotted in a listener are in fact a single command that is parameterized differently following the source interactive object. The following code excerpt, from *ArgoUML*, illustrates this case. Two commands are detected by our algorithm (Lines 9 and 11). These two commands seem to form a single command that takes a parameter, here the string parameter of *setText*.

```
1  public void stateChanged(ChangeEvent ce) {
2    JSlider srcSlider = (JSlider) ce.getSource();
3    Goal d = (Goal) slidersToDecisions.get(srcSlider);
4    JLabel valLab = (JLabel) slidersToDigits.get(srcSlider);
5    int pri = srcSlider.getValue();
6    d.setPriority(pri);
7
8    if (pri == 0) {
9      valLab.setText(Translator.localize("label.off"));
10   } else {
11     valLab.setText("     " + pri);
12   }
13 }
```

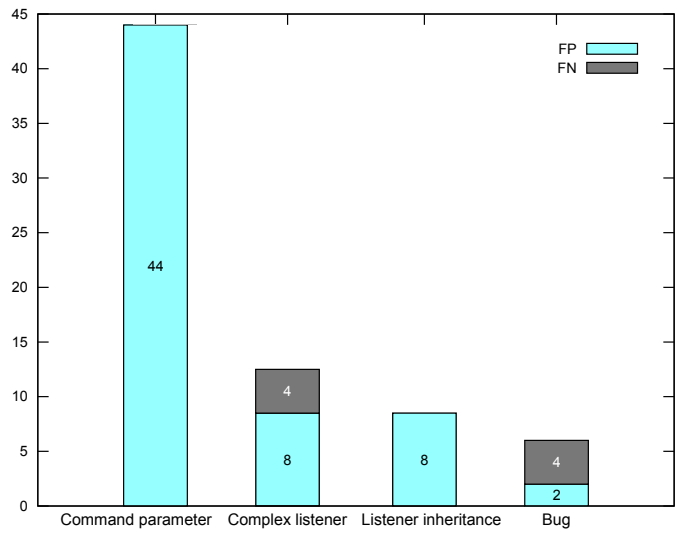

Figure 6: Distribution of the false negative and positive commands

We marked 12 commands are being part of *complex listeners*. A complex listener involves complex and deeply nested conditional statements so that the algorithm fails to detect the commands properly. This category could be part of the bug category since it is a limit of `InspectorGuidget` but we wanted to show the limit of the proposal on complex conditional statements.

Eight listeners have been put in the *listener inheritance* category. This case refers to interactive objects checked several times in the listener inheritance hierarchy. The following code excerpt illustrates this case. Two commands (Lines 10 and 19) are detected by our algorithm for the UI listener of *Panel1*. In fact, these two commands refer to the same interactive object *ok* and the execution of the first command (Line 19) prevents the execution of the second one (Line 10). We consider this case as a design issue of the code.

```
1  public class Panel0 implements ActionListener {
2    Button ok;
3
4    public Panel0() {
5      ok = new JButton();
6      ok.setActionCommand("OK");
7    }
8
9    public void actionPerformed(ActionEvent ae) {
10     if(ae.getActionCommand().equals("OK")) {
11       //...
12     }
13     //...
14   }
15 }
16
17 public class Panel1 extends Panel0 {
18   public void actionPerformed(ActionEvent ae) {
19     if(ae.getActionCommand().equals("OK")) {
20      //...
21      return;
22     }
23     super.actionPerformed(ae);
24   }
25 }
```

Finally, the *bug* category refers to various errors in `InspectorGuidget`.

To conclude on RQ4, our approach is efficient for detecting UI commands that compose UI listener, even if improvements still possible.

RQ5: Blob Listeners Detection Accuracy. To validate that the refactoring is behavior-preserving, the refactored software systems have been manually tested by their developers we contacted and ourselves. Test suites of each system have also been used. Table 4 gives an overview of the results of the *Blob listeners* detection per software system. 12 false positives and one false negative have been identified against 52 *Blob listeners* correctly detected. The average recall is 98.11 % and the average precision is 81.25 %. The average time (computed on five executions for each software system) spent to analyze the software systems is 5.9 s. It excludes the time that *Spoon* takes to load all the classes, that is an average of 22.4 s per software system. We did not consider the time that *Spoon* takes since it is independent of our approach.

Table 4: *Blob listener* detection results

Software System	Successfully Detected *Blob listeners* (#)	FN (#)	FP (#)	$\text{Recall}_{\text{blob}}$ (%)	$\text{Precision}_{\text{blob}}$ (%)	Time (ms)
Eclipse	16	0	2	100	88.89	4
JabRef	8	0	3	100	72.73	5.6
ArgoUML	13	1	2	92.86	86.7	8.6
FreeCol	15	0	5	100	75	5.5
OVERALL	52	1	12	98.11	81.25	5.9

The FP and FN *Blob listeners* is directly linked to the FP and FN of the commands detection. For example, FP commands increased the number of commands in their listener to two or more so that such a listener is wrongly considered as a *Blob listener*. This is the case for *FreeCol* where 47 FP commands led to 5 FP *Blob listeners*.

To conclude on RQ5, regarding the recall and the precision, our approach is efficient for detecting *Blob listeners*.

RQ6: Blob Listeners Refactoring Accuracy.

Table 5: *Blob listener* refactoring results

Software System	Successfully Refactored *Blob listeners* (#)	Failures (#)	$\text{Precision}_{\text{refact}}$ (%)	Time (s)
Eclipse	4	12	25	133
JabRef	4	4	50	236
ArgoUML	11	3	78.57	116
FreeCol	7	8	46.7	135
OVERALL	26	27	49.06	155

This research question aims to provide quantitative results regarding the refactoring of *Blob listeners*. The results of `InspectorGuidget` on the four software systems are described in Table 5. The average refactoring time (*i.e.,* five executions for each software system) is 155 s. Most of the time is spent in finding interactive objects in the code and their usages. We think that optimizations can be done to improve this average time. The average rate of *Blob listeners* successfully refactored using the technique proposed in Section 4 is 55.1 %. 27 of the 49 *Blob listeners* have been refactored. Two main reasons explain this result: 1/ There exists in fact two types of *Blob listeners* and our refactoring solution supports one of them; 2/ The second type of *Blob listeners* may not be refactorable because of limitations of the Java GUI toolkits. Table 6 details the results according to the type of the *Blob listeners*. 86.7 % of the *Blob listeners* that manage several listeners can be refactored. The four failures of this type are related to bugs in the process.

Table 6: Refactoring results considering the type of the *Blob listeners*

Listener Type	Successes (#)	Failures (#)
Several interactive objects for several commands (*e.g.,* action listeners)	26	4
One interactive object for several commands (*e.g.,* mouse and key listeners)	0	23

A *Blob listener* is a listener that can produce several UI commands. The proposed refactoring process works when several interactive objects register the same listener to produce several commands (one command per interactive object). However, in several cases a single interactive object registers a listener to produce several commands. The following code excerpt, simplified from *Jabref*, illustrates this case. A single interactive object registers this listener that treats keyboard events. Several commands, however, are produced in this listener (Lines 6 and 9). Our refactoring solution cannot be applied on such listeners.

```
1  public void keyPressed(KeyEvent e) {
2    //...
3    if (e.isControlDown()) {
4      switch (e.getKeyCode()) {
5        case KeyEvent.VK_UP:
6          frame.getGroupSelector().moveNodeUp(node);
7          break;
8        case KeyEvent.VK_DOWN:
9          frame.getGroupSelector().moveNodeDown(node);
10         break;
11       //...
12 }
```

Refactoring solutions for such key listeners may exist for several GUI toolkits that support key bindings. For example, in the following code we manually fragmented the initial listener two atomic Java Swing key bindings (Lines 4 and 8). Such a refactoring strongly depends on the targeted UI toolkit.

```
1  InputMap im = textfield.getInputMap();
2  ActionMap a = textfield.getActionMap();
3
4  im.put(KeyStroke.getKeyStroke(KeyEvent.VK_UP,
5         InputEvent.CTRL_MASK), "up");
6  a.put("up", e -> frame.getGroupSelector().moveNodeUp(node));
7
8  im.put(KeyStroke.getKeyStroke(KeyEvent.VK_DOWN,
9         InputEvent.CTRL_MASK), "down");
10 a.put("down",e->frame.getGroupSelector().moveNodeDown(node));
```

Another example of listeners that cannot be refactored is depicted in the following code excerpt. This mouse listener contains three commands (Lines 3, 5, and 7). Similarly than

Table 7: Commits and discussions

Software System	Bug reports, commits, and discussions
Eclipse (platform.ui)	https://bugs.eclipse.org/bugs/show_bug.cgi?id=510745
	https://dev.eclipse.org/mhonarc/lists/platform-ui-dev/msg07651.html
JabRef	https://github.com/JabRef/jabref/pull/2369/
	https://github.com/JabRef/jabref/commit/021f094e64a6393a7490ee680d73ef26b3128625
ArgoUML	http://argouml.tigris.org/issues/show_bug.cgi?id=6524
FreeCol	https://sourceforge.net/p/freecol/mailman/message/35566820/
	https://sourceforge.net/p/freecol/git/ci/669cf9c74b208c141cea27ee254292b3422d5718/
	https://sourceforge.net/p/freecol/git/ci/2865215d3712a8d4d4bd958c1b397c90460192da/
	https://sourceforge.net/p/freecol/git/ci/cdc689c7ae4bbac9fcc729477d5cc7e40ac4a90b/
	https://sourceforge.net/p/freecol/git/ci/0eedd71b269b6cf20ec00f0fc5a7da932ceaab4f/
	https://sourceforge.net/p/freecol/git/ci/973422623b52289481f328b27f12543a4b383f38/
	https://sourceforge.net/p/freecol/git/ci/985adc4de11ccd33648e99294e5d91319cb23aa0/
	https://sourceforge.net/p/freecol/git/ci/4fe44e747cb30a161d8657750aa75b6c57ea30ab/

for the previous key listener, our refactoring solution cannot be applied on such listeners. Moreover, to the best of our knowledge no GUI toolkit permits the definition of mouse bindings as in the previous code example. Such a *Blob listener* cannot thus be refactored.

```
public void mousePressed(MouseEvent e) {
  if (e.getClickCount() == 1) {
    // ...
  }else if (e.getButton() == MouseEvent.BUTTON3) {
    //...
  } else if (e.getClickCount() > 1) {
    //...
  }
}
```

To conclude on RQ6, the refactoring solution we propose is efficient for one of the two types of *Blob listeners*. Refactoring the second type of *Blob listeners* may not be possible and depends on the targeted GUI toolkit.

RQ7: Relevance of the *Blob listener*.

This last research question aims to provide qualitative results regarding the refactoring of *Blob listeners*. We computed code metrics before and after the refactoring to measure the impact of the changes on the code. We also submitted patches that remove the found and refactorable *Blob listeners* from the analyzed software systems. We then asked their developers for feedback regarding the patches and the concept of *Blob listener*. The bug reports that contain the patches, the commits that remove *Blob listeners*, and the discussions are listed in Table 7. Once the refactored code automatically produced using `InspectorGuidget`, we manually applied some changes to follow the coding conventions of the different software systems. Then, the patches have been manually created by applying a *diff* between the original code and the automatically refactored one. The patches submitted to *Jabref* and *FreeCol* have accepted and merged. The patches for Eclipse are not yet merged but were positively commented. We did not receive any comment regarding the patches for *ArgoUML*. We noticed that *ArgoUML* is no more actively maintained.

We asked developers whether they consider that coding UI listeners that manage several interactive objects is a bad coding practice. The developers that responded globally agree that *Blob listener* is a design smell. "*It does not strictly violate the MVC pattern. [...] Overall, I like your solution*". "*Probably yes, it depends, and in examples you've patched this was definitely a mess*". An Eclipse developer suggest to complete the Eclipse UI development documentation to add information related to UI design smells and *Blob listener*.

Regarding the relevance of the refactoring solution: "*I like it when the code for defining a UI element and the code for interacting with it are close together. So hauling code out of the action listener routine and into a lambda next to the point a button is defined is an obvious win for me.*" A developer, however, explained that "*there might be situations where this can not be achieved fully, e.g. due to limiting implementations provided by the framework.*" We agree with this statement and the outcomes of RQ6 explain the limitations. "*It depends, if you refactor it by introducing duplicated code, then this is not suitable and even worse as before*". We also agree with this statement that we commented by computing several code metrics before and after the refactoring, as summarized in Table 8. We computed the changes in terms of number of lines of code (LoC), cyclomatic complexity (CC), and duplicated lines (DUP). The refactoring of *Blob listeners* reduces the number of lines of code (-210 LoCs) and the cyclomatic complexity (-150 CC). We noticed 25 duplicated lines of code. These duplicated lines are pre- and post-statements of commands (see Figure 1), *e.g.,* variable declarations, used by several commands in the same listener. These statements are thus duplicated when the listener is separated into atomic ones.

Table 8: Code metrics changes with *Blob listeners* refactoring

Software System	LoC (#)	CC (#)	DUP (#)
Eclipse	−40	−45	11
JabRef	−49	−21	0
ArgoUML	−35	−47	13
FreeCol	−146	−37	1
OVERALL	−270	−150	25

To conclude on RQ7, the concept of *Blob listener* and the refactoring solution we propose is accepted by the developers we interviewed. The refactoring has a positive impact on the code quality. The interviews did not permit the identification of how *Blob listener* are introduced in the code, following the classification of Sharma *et al.* [25].

5.4. Threats to validity

External validity. This threat concerns the possibility to generalize our findings. We designed the experiments using multiple Java Swing and SWT open-source software systems to diversify the observations. These unrelated software systems are developed by different persons and cover various user interactions. Our implementation and our empirical study (Section 3) focus on the Java Swing and SWT toolkits. We focused on the Java Swing and SWT toolkits because of their popularity and the large quantity of Java Swing and SWT legacy code. We provide on the companion web page examples of *Blob listeners* in other Java UI toolkits, namely GWT and JavaFX[2].

Construct validity. This threat relates to the perceived overall validity of the experiments. Regarding the empirical study (Section 3), we used `InspectorGuidget` to find UI commands in the code. `InspectorGuidget` might not have detected all the UI commands. We showed in the validation of this tool (Section 5) that its precision (95.7) and recall (99.49) limit this threat. Regarding the validation of `InspectorGuidget`, the detection of FNs and FPs have required a manual analysis of all the UI listeners of the software systems. To limit errors during this manual analysis, we added a debugging feature in `InspectorGuidget` for highlighting UI listeners in the code. We used this feature to browse all the UI 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 UI listener detection. We also manually determined whether a listener is a *Blob listener*. To reduce this threat, we carefully inspected each UI command highlighted by our tool.

5.5. Scope of the Approach

Our approach has the following limitations. First, the command detection algorithm can be improved by detecting command parameters, *i.e.,* commands in a same listener that form a single command that can be parameterized. Second, the refactoring of key listeners, using key bindings, can be supported by `InspectorGuidget`. This support may vary and depend on the targeted UI toolkit.

The current implementation of the approach supports JavaFX, Swing, and SWT code. The approach is, however, not limited to these graphical toolkits and Java. The concept of UI listener (at the origin of the Blob Listener) is used, to the best of our knowledge, by most of the modern graphical libraries usually along with data binding.

6. Related Work

Work related to this paper fall into three categories: design smell detection; UI maintenance and evolution; UI testing.

6.1. Design Smell Detection

The characterization and detection of object-oriented (OO) design smells have been widely studied [25, 28]. For instance, research works characterized various OO design smells associated with code refactoring operations [11, 29]. Multiple empirical studies have been conducted to observe the impact of several OO design smells on the code. These studies show that OO design smells can have a negative impact on maintainability [30], understandability [20], and change- or fault-proneness [6]. While developing seminal advances on OO design smells, these research works focus on OO concerns only. Improving the validation and maintenance of UI code implies a research focus on UI design smells, as we propose in this paper.

Related to UI code analysis, Table 9 summarizes several UI design smells discussed in the next paragraphs. Surveys and classifications [25, 28] on OO design smells identified characteristics of code smells that can be applied to UI design smells. We used several of these characteristics to describe the UI design smells of Table 9: *focus*: the focus area where a design smell operates [25]; *structural principle*: what structural principles a design smell violates [26]; *detection strategy*: the techniques used to detect a design smell [25]. The *Blob listener* focuses on the implementation of controllers. Interviews with developers and code exhibited that a problem of encapsulation: UI listeners may be visible as super interfaces or classes of controllers. We use a metric and code analysis techniques to detect *Blob listener* instances.

Aniche *et al.* define several design smells that affect web applications [9]. In particular, a design smell focuses on web controllers that bind the model (on the server side) and the UI (on the client side) of the application. They define six web design smells. Four of them concern the model component and are thus not related to UI. The two remaining smells concern web controllers, *i.e.,* the server-side objects that receives queries from the client side. The *Promiscuous Controller* smell refers to *a [web] controller offering too many actions*. The detection of such a controller is based on the number of web routes (10) and services (3) it contains. The *Brain Controller* smell refers to a lack of separation of concerns in a web controller so that it becomes too complex. These two design smells refer to implementation and design issues, and a weakness in the modularity of web controllers. Metrics are used to identified these design smells in the code. Web controllers and UI controllers strongly differ (web routes and services *vs* interactive objects and UI listeners) so that our code analyses cannot be compared. However, their results regarding promiscuous controllers follow ours on *Blob listeners*: web or UI controllers should not do too much.

Silva *et al.* propose an approach to inspect UI source code as a reverse engineering process [31, 32]. Their goal is to provide developers with a framework supporting the development of UI metrics and code analyses. They also applied standard OO code metrics on UI code [33]. Closely, Almeida *et al.* propose a set of UI smells that focus on usability [27]. These smells are described in Table 9. Several of them (*UI Shotgun Surgery*, *UI Middle Man*, *UI Inappropriate Intimacy*, and *UI Feature Envy*) are adaptations of the object-oriented design smells of

Table 9: UI design smells proposed in related works

Name	Description	Focus [25]	Structural Principle [26]	Detection Strategy [25]
Blob Listener	**A UI listener/handler that controls too much interactive objects**	Implementation	Deficient encapsulation	Metric-based
Promiscuous Controller [9]	web server-side controllers that manage too many routes	Design and Implementation	Weakened modularity	Metric-based
Brain Controller [9]	Server-side controllers that do too much (lack of separation of concern)	Design and Implementation	Weakened modularity	Metric-based
UI Shotgun Surgery [27]	A change on the UI structure spans over multiple UI implementations	Usability	N/A	None
Too Many Layers [27]	A UI is composed of too many layers (*e.g.,* windows)	Usability	N/A	None
UI Middle Man [27]	A UI component (*e.g.,* a window) delegates the job to another UI component	Usability	N/A	None
Information Overload [27]	Too much information is provided to users	Usability	N/A	None
UI Inappropriate Intimacy [27]	Several UIs, accessible from different places, handle related domain elements	Usability	N/A	None
UI Feature Envy [27]	One UI allows user to perform a task also provided by another UI of the system	Usability	N/A	None

the same name [11] with a focus on UI code. The *Too Many Layers* and *Information Overload* UI design smells are related to the structural complexity of UIs that may have a negative impact on their understanding by users. The relevance of these UI smells are not studied by the authors. These smells, however, show that UIs are software artifacts that require specific analysis techniques to improve their quality.

The automatic detection of design smells involves two steps. First, a source code analysis is required to compute source code metrics. Second, heuristics are applied to detect design smells on the basis of the computed metrics to detect design smells. Source code analyses can take various forms, notably: static, as we propose, and historical. Regarding historical analysis, Palomba *et al.* propose an approach to detect design smells based on change history information [5]. Future work may also investigate whether analyzing code changes over time can help in characterizing *Blob listeners*. Regarding detection heuristics, the use of code metrics to define detection rules is a mainstream technique. Metrics can be assemble with threshold values defined empirically to form detection rules [34]. Search-based techniques are also used to exploit OO code metrics [35], as well as machine learning [36], or bayesian networks [37]. Still, these works do not cover UI design smells. In this paper, we focus on static code analysis to detect UI commands to form a *Blob listener* detection rule. To do so, we used a Java source code analysis framework that permits the creation of specific code analyzers [19]. Future work may investigate other heuristics and analyses to detect UI design smells.

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 [38]. Garcia *et al.* propose an approach for identifying architectural design smells [39]. Similarly, this work aims at motivating that UIs form another domain concerned by specific design smells that have to be characterized.

Research studies have been conducted to evaluate the impact of design smells on system's quality [40, 41] or how they are perceived by developers [23]. Future work may focus on how software developers perceive *Blob listeners*.

6.2. UI maintenance and evolution

Unlike OO design smells, less research work focuses on UI design smells. Zhang *et al.* propose a technique to automatically repair broken workflows in Swing UIs [42]. Static analyses are proposed. This work highlights the difficulty "*for a static analysis to distinguish UI actions [UI commands] that share the same event handler [UI listener]*". In our work, we propose an approach to accurately detect UI commands that compose UI listeners. Staiger also proposes a static analysis to extract UI code, interactive objects, and their hierarchies in C/C++ software systems [43]. The approach, however, is limited to find relationships between UI elements and thus does not analyze UI controllers and their listeners. Zhang *et al.* propose a static analysis to find violations in UIs [44]. These violations occur when UI operations are invoked by non-UI threads leading a UI error. The static analysis is applied to infer a static call graph and check the violations. Frolin *et al.* propose an approach to automatically find inconsistencies in MVC JavaScript applications [18]. UI 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.

6.3. UI testing

UI testing may also implies UI code analysis techniques. The automatic production of UI tests requires techniques to extract UI information at design or run time. Such techniques may involve the dynamic detection of interactive objects to automatically interact with them at run time [45]. Several UI testing techniques improve the automatic interaction with UIs by analyzing in the code the dependencies between the interactive objects and the UI listeners [46, 47, 48, 49]. UI listeners are analyzed to: identify class fields shared by several listeners [46]; detect dependencies between UI listeners [47]; detect transitions between graphical windows [48]; identify, from one-command listeners, relevant input data to test interactive objects and produce UI tests [49].

7. Conclusion and Future Work

7.1. Conclusion

In this paper, we investigated a new research area on UI design smells. We detailed a specific UI design smell, we called *Blob listener*, that can affect UI listeners. The empirical study we conducted exhibits a specific number of UI commands per UI listener that characterizes a *Blob listener* exists. We defined this threshold to three UI commands per UI listener. We detailed an algorithm to automatically detect *Blob listeners*. We then proposed a behavior-preserving algorithm to refactor detected *Blob listeners*. The detection and refactoring algorithms have been implemented in a tool publicly available and then evaluated. The results showed that the *Blob listeners* detection and refactoring techniques are effective with several possible improvements. Developers agreed that the *Blob listeners* is a design smell.

7.2. Research Agenda

UI code is a major part of the code of software systems more and more interactive. We argue that code analysis tools, similar to *Findbugs* or *PMD* that analyze object-oriented code, should be developed to specifically detect UI design smells. `InspectorGuidget` is a first step in this way. In our future work, we first aim to complete `InspectorGuidget` with other UI design smells that we would empirically identify and characterize. We will investigate whether some UI faults [50] are accentuated by UI design smells. Second, the current UI event handling, based on the observer pattern, faces several critical limits [51]. UI event handling, however, is still used by a majority of modern UI libraries along with data binding mechanisms. We think that UI event handling should be replaced by reactive programming / data binding approaches specifically designed to react on UI events.

Acknowledgments

We thank Yann-Gaël Guéhéneuc for his insightful comments on this paper. We also thank Jean-Rémy Falleri for his help on the statistical analysis.

References

[1] G. E. Krasner, S. T. Pope, A description of the Model-View-Controller user interface paradigm in Smalltalk80 system, Journal of Object Oriented Programming 1 (1988) 26–49.

[2] M. Potel, MVP: Model-View-Presenter the Taligent programming model for C++ and Java, Taligent Inc.

[3] B. 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, 1991, pp. 211–220. doi:10.1145/120782.120805.

[4] J. Smith, WPF apps with the model-view-viewmodel design pattern, MSDN Magazine.
URL http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

[5] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, Software Engineering, IEEE Transactions ondoi:10.1109/TSE.2014.2372760.

[6] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Software Engineering 17 (3) (2012) 243–275. doi:10.1007/s10664-011-9171-y.

[7] A. Lozano, M. Wermelinger, B. Nuseibeh, Assessing the impact of bad smells using historical information, in: Workshop on Principles of software evolution, ACM, 2007, pp. 31–34. doi:10.1145/1294948.1294957.

[8] D. Rapu, S. Ducasse, T. Gîrba, R. Marinescu, Using history information to improve design flaws detection, in: Proc. of Conference on Software Maintenance and Reengineering, 2004, pp. 223–232. doi:10.1109/CSMR.2004.1281423.

[9] M. Aniche, G. Bavota, C. Treude, M. A. Gerosa, A. van Deursen, Code smells for model-view-controller architectures, Empirical Software Engineering (2017) 1–37doi:10.1007/s10664-017-9540-2.

[10] V. Lelli, A. Blouin, B. Baudry, F. Coulon, O. Beaudoux, Automatic detection of GUI design smells: The case of blob listener, in: EICS'16: Proceedings of the 8th ACM SIGCHI symposium on Engineering interactive computing systems (EICS 2016), ACM, 2016.
URL https://hal.inria.fr/hal-01308625

[11] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Boston, 1999.

[12] W3C, MBUI - abstract user interface models, Tech. rep. (2014).
URL https://www.w3.org/TR/abstract-ui/

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley, 1995.

[14] M. Beaudouin-Lafon, Instrumental interaction: an interaction model for designing post-WIMP user interfaces, in: Proc. of CHI'00, ACM, 2000, pp. 446–453.
URL 10.1145/332040.332473

[15] A. Blouin, O. Beaudoux, Improving modularity and usability of interactive systems with Malai, in: Proc. of EICS'10, 2010, pp. 115–124.
URL https://hal.inria.fr/inria-00477627

[16] A. Blouin, B. Morin, G. Nain, O. Beaudoux, P. Albers, J.-M. Jézéquel, Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation, in: EICS'11: Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems, 2011, pp. 85–94. doi:10.1145/1996461.1996500.

[17] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, SIGSOFT Softw. Eng. Notes 30 (2005) 1–36. doi:10.1145/1050849.1050865.

[18] F. Ocariza, K. Pattabiraman, A. Mesbah, Detecting inconsistencies in JavaScript MVC applications, in: Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE), ACM, 2015, p. 11 pages. doi:10.1109/ICSE.2015.52.

[19] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, L. Seinturier, SPOON: A library for implementing analyses and transformations of Java source code, Software: Practice and Experience 43 (4). doi:10.1002/spe.2346.

[20] M. Abbes, F. Khomh, Y. G. Guéhéneuc, G. Antoniol, An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension, in: Proceedings of the European Conference on Software Maintenance and Reengineering, 2011, pp. 181–190. doi:10.1109/CSMR.2011.24.

[21] D. J. Sheskin, Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition, Chapman & Hall/CRC, 2007.

[22] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs?, 2013, pp. 672–681. doi:10.1109/ICSE.2013.6606613.

[23] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: Proc. of ICSM'14, IEEE, 2014, pp. 101–110. doi:10.1109/ICSME.2014.32.

[24] T. Mens, T. Tourwé, A Survey of Software Refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139. doi:10.1109/TSE.2004.1265817.

[25] T. Sharma, D. Spinellis, A survey on software smells, Journal of Systems and Software 138 (2018) 158 – 173. doi:10.1016/j.jss.2017.12.034.

[26] S. Ganesh, T. Sharma, G. Suryanarayana, Towards a principle-based classification of structural design smells., Journal of Object Technology 12 (2) (2013) 1–1.
URL http://www.jot.fm/issues/issue_2013_06/article1.pdf

[27] D. Almeida, J. C. Campos, J. a. Saraiva, J. a. C. Silva, Towards a catalog of usability smells, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, ACM, 2015, pp. 175–181. doi:10.1145/2695664.2695670.

[28] G. Rasool, Z. Arshad, A review of code smell mining techniques, Journal of Software: Evolution and Process 27 (11) (2015) 867–895. doi:10.1002/smr.1737.

[29] W. J. Brown, H. W. McCormick, T. J. Mowbray, R. C. Malveau, AntiPatterns: refactoring software, architectures, and projects in crisis, Wiley New York, 1998.

[30] A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: an empirical study, in: 35th International Conference on Software Engineering, ICSE '13, 2013, pp. 682–691. doi:10.1109/ICSE.2013.6606614.

[31] J. C. Silva, J. C. Campos, J. A. Saraiva, GUI inspection from source code analysis.
URL http://hdl.handle.net/1822/18517

[32] J. a. C. Silva, C. Silva, R. D. Gonçalo, J. a. Saraiva, J. C. 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, ACM, 2010, pp. 181–186. doi:10.1145/1822018.1822045.

[33] J. C. Silva, J. C. Campos, J. Saraiva, J. L. Silva, An approach for graphical user interface external bad smells detection, in: New Perspectives in Information Systems and Technologies, 2014, pp. 199–205. doi:10.1007/978-3-319-05948-8_19.

[34] N. Moha, Y.-G. Gueheneuc, L. Duchien, A. Le Meur, DECOR: A method for the specification and detection of code and design smells, Software Engineering, IEEE Transactions on 36 (1) (2010) 20–36. doi:10.1109/TSE.2009.50.

[35] D. Sahin, M. Kessentini, S. Bechikh, K. Deb, Code-smell detection as a bilevel problem, ACM Trans. Softw. Eng. Methodol. 24 (1) (2014) 6:1–6:44. doi:10.1145/2675067.

[36] M. Zanoni, F. A. Fontana, F. Stella, On applying machine learning techniques for design pattern detection, Journal of Systems and Software 103 (0) (2015) 102 – 117. doi:10.1016/j.jss.2015.01.037.

[37] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, BDTEX: A GQM-based bayesian approach for the detection of antipatterns, Journal of Systems and Software 84 (4) (2011) 559–572. doi:doi.org/10.1016/j.jss.2010.11.921.

[38] N. Moha, F. Palma, M. Nayrolles, B. Joyen Conseil, G. Yann-Gael, B. Baudry, J.-M. Jézéquel, Specification and Detection of SOA Antipatterns, in: International Conference on Service Oriented Computing, 2012.
URL https://hal.inria.fr/hal-00722472

[39] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on, IEEE, 2009, pp. 255–258. doi:10.1109/CSMR.2009.59.

[40] S. M. Olbrich, D. S. Cruzes, D. I. Sjoberg, Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems, in: 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10. doi:10.1109/ICSM.2010.5609564.

[41] F. A. Fontana, V. Ferme, A. Marino, B. Walter, P. Martenka, Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains, in: Proc. of ICSM'13, IEEE, 2013, pp. 260–269. doi:10.1109/ICSM.2013.37.

[42] S. Zhang, H. Lü, M. D. Ernst, Automatically repairing broken workflows for evolving GUI applications, in: ISSTA 2013, Proceedings of the 2013 International on Software Testing and Analysis, 2013, pp. 45–55. doi:10.1145/2483760.2483775.

[43] S. Staiger, Static analysis of programs with graphical user interface, in: Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on, 2007, pp. 252–264. doi:10.1109/CSMR.2007.44.

[44] S. Zhang, H. Lü, M. D. Ernst, Finding errors in multithreaded GUI applications, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, ACM, 2012, pp. 243–253. doi:10.1145/2338965.2336782.

[45] B. N. Nguyen, B. Robbins, I. Banerjee, A. Memon, GUITAR: an innovative tool for automated testing of GUI-driven software, Automated Software Engineering 21 (1) (2014) 65–105. doi:10.1007/s10515-013-0128-9.

[46] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, A. M. Memon, Lightweight static analysis for GUI testing, in: Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on, IEEE, 2012, pp. 301–310. doi:10.1109/ISSRE.2012.25.

[47] S. Yang, D. Yan, H. Wu, Y. Wang, A. Rountev, Static control-flow analysis of user-driven callbacks in android applications, in: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, Vol. 1, IEEE, 2015, pp. 89–99. doi:10.1109/ICSE.2015.31.

[48] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, A. Rountev, Static window transition graphs for android (t), in: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, 2015, pp. 658–668. doi:10.1109/ASE.2015.76.

[49] S. Ganov, C. Killmar, S. Khurshid, D. E. Perry, Event listener analysis and symbolic execution for testing GUI applications, in: International Conference on Formal Engineering Methods, Springer, 2009, pp. 69–87. doi:10.1007/978-3-642-10373-5_4.

[50] V. Lelli, A. Blouin, B. Baudry, Classifying and qualifying GUI defects, in: IEEE International Conference on Software Testing, Verification and Validation (ICST 2015), IEEE, 2015.
URL https://hal.inria.fr/hal-01114724v1

[51] I. Maier, M. Odersky, Deprecating the observer pattern with scala. react, Tech. rep. (2012).
URL https://infoscience.epfl.ch/record/176887