Automatic Detection of GUI Design Smells: The Case of Blob Listener

Valéria Lelli, Arnaud Blouin, Benoit Baudry, Fabien Coulon, Olivier Beaudoux

Automatic Detection of GUI Design Smells: The Case of Blob Listener

Valéria Lelli
University of Ceará, Brazil
valerialelli@great.ufc.br

Arnaud Blouin
INSA Rennes, France
arnaud.blouin@irisa.fr

Benoit Baudry
Inria, France
benoit.baudry@inria.fr

Fabien Coulon
Inria, France
fabien.coulon@inria.fr

Olivier Beaudoux
ESEO, France
olivier.beaudoux@eseo.fr

ABSTRACT

Graphical User Interfaces (GUIs) intensively rely on event-driven programming: widgets send GUI events, which capture users' interactions, to dedicated objects called *controllers*. Controllers implement several *GUI listeners* that handle these events to produce GUI commands. In this work, we conducted an empirical study on 13 large Java Swing open-source software systems. We study to what extent the number of GUI commands that a GUI listener can produce has an impact on the change- and fault-proneness of the GUI listener code. We identify a new type of design smell, called *Blob listener* that characterizes GUI listeners that can produce more than two GUI commands. We show that 21 % of the analyzed GUI controllers are *Blob listeners*. We propose a systematic static code analysis procedure that searches for *Blob listener* that we implement in `InspectorGuidget`. We conducted experiments on six software systems for which we manually identified 37 instances of *Blob listener*. `InspectorGuidget` successfully detected 36 *Blob listeners* out of 37. The results exhibit a precision of 97.37 % and a recall of 97.59 %. Finally, we propose coding practices to avoid the use of *Blob listeners*.

Author Keywords

User interface, design smell, software validation, code quality

ACM Classification Keywords

F.3.3 Studies of Program Constructs: Control primitives; H.5.2 User Interfaces: Graphical user interfaces (GUI); D.3.3 Language Constructs and Features: Patterns

INTRODUCTION

Graphical User Interfaces (GUI) are the visible and tangible vector that enable users to interact with software systems. While GUI design and qualitative assessment is handled by GUI designers, integrating GUIs into software systems remains a software engineering task. Software engineers develop GUIs following widespread architectural design patterns, such as MVC [18] or MVP [30] (*Model-View-Controller/Presenter*), that consider GUIs as first-class concerns (*e.g.,* the *View* in these two patterns). These patterns clarify the implementations of GUIs by clearly separating concerns, thus minimizing the "spaghetti of call-backs" [23]. These implementations rely on event-driven programming where events are treated by *controllers* (resp. presenters[1]), as depicted by Listing 1. In this code example, the *AController* controller manages three widgets, *b1*, *b2*, and *m3* (Lines 2–4). To handle events that these widgets trigger in response to users' interactions, the GUI listener *ActionListener* is implemented in the controller (Lines 6–17). One major job of GUI listeners is the production of GUI commands, *i.e.,* a set of statements executed in reaction of a GUI event produced by a widget (Lines 9, 11, and 15). Like any code artifact, GUI controllers must be tested, maintained and are prone to evolution and errors. In particular, software developers are free to develop GUI listeners that can produce a single or multiple GUI commands. In this work, we investigate the effects of such development practices on the code quality of the GUI listeners.

```
1  class AController implements ActionListener {
2    JButton b1;
3    JButton b2;
4    JMenuItem m3;
5
6    @Override public void actionPerformed(ActionEvent e) {
7       Object src = e.getSource();
8       if(src==b1){
9          // Command 1
10      }else if(src==b2)
11         // Command 2
12      }else if(src instanceof AbstractButton &&
13           ((AbstractButton)src).getActionCommand().equals(
14           m3.getActionCommand()))
15         // Command 3
16      }
17 }}
```

Listing 1. Code example of a GUI controller

In many cases GUI code is intertwined with the rest of the code. We thus propose a static code analysis required for detecting the GUI commands that a GUI listener can produce. Using this code analysis, we then conduct a large empirical study on Java Swing open-source GUIs. We focus on the Java Swing toolkit because of its popularity and the large quantity of Java Swing legacy code. We empirically study to what extent the

[1]For simplicity, we use the term *controller* to refer to any kind of component of MV* architectures that manages events triggered by GUIs, such as *Presenter* (MVP), or *ViewModel* (MVVM [37]).

number of GUI commands that a GUI listener can produce has an impact on the change- or fault-proneness of the GUI listener code, considered in the literature as negative impacts of a design smell on the code [27, 16, 20, 31]. Based on the results of this experiment, we define a GUI design smell we call *Blob listener*, *i.e.,* a GUI listener that can produce more than two GUI commands. For example with Listing 1, the GUI listener implemented in *AController* manages events produced by three widgets, *b1*, *b2*, and *m3* (Lines 8, 10, and 13), that produce one GUI command each. 21% of the analyzed GUI controllers are *Blob listeners*.

We provide an open-source tool, `InspectorGuidget`[2], that automatically detect *Blob listeners* in Java Swing GUIs. To evaluate the ability of `InspectorGuidget` at detecting *Blob listeners*, we considered six representative Java software systems. We manually retrieved all instances of *Blob listener* in each application, to build a ground truth for our experiments: we found 37 *Blob listeners*. `InspectorGuidget` detected 36 *Blob listeners* out of 37. The experiments show that our algorithm has a precision of 97.37 % and recall of 97.59 % to detect *Blob listeners*. Our contributions are:

- an empirical study on 13 Java Swing open-source software systems. This study investigates the current coding practices of GUI controllers. The main result of this study is the identification of a GUI design smell we called *Blob listener*.
- a precise characterization of the *Blob listener*. We also discuss the different coding practices of GUI listeners we observed in listeners having less than three commands.
- an open-source tool, `InspectorGuidget`, that embeds a static code analysis to automatically detect the presence of *Blob listeners* in Swing GUIs. We evaluated the ability of `InspectorGuidget` at detecting *Blob listeners*.

The paper is organized as follows. Section 2 describes an empirical study that investigates coding practices of GUI controllers. Based on this study, Section 3 describes an original GUI design smell we called *Blob listener*. Following, Section 4 introduces an algorithm to detect *Blob listeners*, evaluated in Section 5. The paper ends with related work (Section 7) and a research agenda (Section 8).

AN EMPIRICAL STUDY ON GUI LISTENERS

All the material of the experiments is freely available on the companion web page[2].

Independent Variables

GUI listeners are core code artifacts in software systems. They receive and treat GUI events produced by users while interacting with GUIs. In reaction of such events, GUI listeners produce GUI commands that can be defined as follows:

Definition 1 (GUI Command) *A GUI command [13, 5], aka.* action *[6, 7], is a set of statements executed in reaction to a user interaction, captured by an input event, performed on a GUI. GUI commands may be supplemented with: a precondition checking whether the command fulfills the prerequisites to be executed; undo and redo functions for, respectively, canceling and re-executing the command.*

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

Number of GUI Commands (**CMD**). This variable measures the number of GUI commands a GUI listener can produce. To measure this variable, we develop a dedicated static code analysis (see Section 4).

GUI listeners are thus in charge of the relations between a software system and its GUI. As any code artifact, GUI code has to be tested and analyzed to provide users with high quality (from a software engineering point of view) GUIs. In this work, we specifically focus on a coding practice that affect the code quality of the GUI listeners: we want to *state whether the number of GUI commands that GUI listeners can produce has an effect on the code quality of these listeners*. Indeed, software developers are free to develop GUI listeners that can produce a single or multiple GUI commands since no coding practices or GUI toolkits provide coding recommendations. To do so, we study to what extent the number of GUI commands that a GUI listener can produce has an impact on the change- and fault-proneness of the GUI listener code. Such a correlation has been already studied to evaluate the impact of several antipatterns on the code quality [16].

We formulate the research questions of this study as follows:

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

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

RQ3 Does a threshold value, *i.e.,* a specific number of GUI commands per GUI listener, that can characterize a GUI design smell exist?

Dependent Variables

To answer the previously introduced research questions, we measure the following dependent variables.

Average Commits (**COMMIT**). This variable measures the average number of commits per line of code (LoC) of GUI listeners. This variable will permit to evaluate the change-proneness of GUI listeners. The measure of this variable implies that the objects of this study, *i.e.,* software systems that will be analyzed, must have a large and accessible change history. To measure this variable, we automatically count the number of the commits that concern each GUI listener.

Average fault Fixes (**FIX**). This variable measures the average number of fault fixes per LoC of GUI listeners. This variable will permit to evaluate the fault-proneness of GUI listeners. The measure of this variable implies that the objects of this study must use a large and accessible issue-tracking system. To measure this variable, we manually analyze the log of the commits that concern each GUI listener. We 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).

Both COMMIT and FIX rely on the ability to get the commits that concern a given GUI listener. For each software system, we use all the commits of their history. To identify the start and end lines of GUI listeners, we developed a static code

analysis. This code analysis uses the definition of the GUI listener methods provided by GUI toolkits (*e.g., void actionPerformed(ActionEvent)*) to locate these methods in the code. Moreover, commits may change the position of GUI listeners in the code (by adding or removing LoCs). To get the exact position of a GUI 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.

Objects

The objects of this study are a set of large 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. Their change history must be large (*i.e.,* must have numerous commits) to let the analysis of the commits relevant. In this work, we focus on Java Swing GUIs because of the popularity and the large quantity of Java Swing legacy code. We thus selected from the Github platform[4] 13 large Java Swing software systems. The average number of commits of these software systems is approximately 2500 commits. The total size of Java code is 1414k Java LoCs. Their average size is approximately 109k Java LoCs.

Results

We can first highlight that the total number of GUI listeners producing at least one GUI command identified by our tool is 858, *i.e.,* an average of 66 GUI listeners per software system. This approximately corresponds to 20 kLoCs, *i.e.,* around 1.33 % of their Java code.

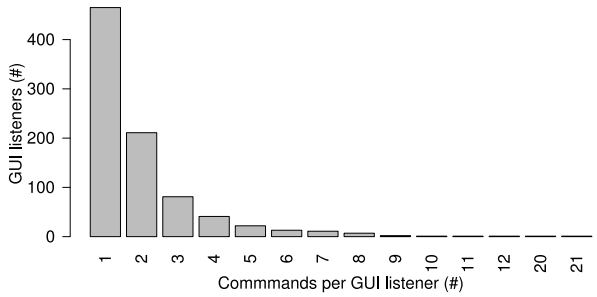

Figure 1. Distribution of the listeners according to their number of GUI commands

Figure 1 shows the distribution of the listeners according to their number of GUI commands. Most of the listeners (465) can produce one command (we will call 1-command listener). 211 listeners can produce two commands. 81 listeners can produce three commands. 101 listeners can produce at least four commands. To obtain representative data results, we will consider in the following analyses four categories of listeners: *one-command listener*, *two-command listener*, *three-command listener*, and *four+-command listener*.

Besides, a first analysis of the data exhibits many outliers, in particular for the one-command listeners. To understand the presence of these outliers, we manually scrutiny some of them and their change history. We observe that some of these outliers are GUI listeners which size has been reduced over the commits. For instance, we identified outliers that contained multiple GUI commands before commits that reduced them as one- or two-command listeners. Such listeners distort the analysis of the results by considering listeners that have been large, as one- or two-command listeners. We thus removed those outliers from the data set, since outliers removal, when justified, may bring benefits to the data analysis [26]. We compute the box plot statistics to identify and then remove the outliers.

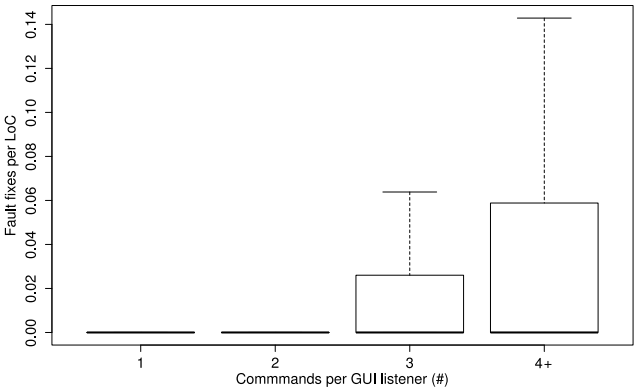

Figure 2. Number of bug fixes per LoC of GUI listeners

Figure 2 depicts the number of fault fixes per LoC (*i.e., FIX*) of the analyzed GUI listeners. We observe an increase of the fault fixes per LoC when $CMD \geq 3$. These results are detailed in Table 1. The mean value of *FIX* constantly increases over *CMD*. Because these data follow a monotonic relationship, we use the Spearman's rank-order correlation coefficient to assess the correlation between the number of fault fixes per LoC and the number of GUI commands in GUI listeners [26]. We also use a 95 % confidence level (*i.e., p*-value$<$0.05). This test exhibits a low correlation (0.4438) statistically significant with a *p*-value of 2.2×10^{-16}.

Table 1. Mean, correlation, and significance of the results

Dependent variables	Mean CMD=1	Mean CMD=2	Mean CMD=3	Mean CMD>3	Correlation	Significance *p*-value
FIX	0	0.0123	0.0190	0.0282	0.4438	<0.001
COMMIT	0.0750	0.0767	0.0849	0.0576	0.0570	0.111

Regarding **RQ1**, on the basis of these results we can conclude that *the number of GUI commands per GUI listeners does not have a strong negative impact on fault-proneness of the GUI listener code.* This result is surprising regarding the global increase that can be observed in Figure 2. One possible explanation is that the mean of the number of bugs per LoC slowly increases over the number of commands as shown in the first row of Table 1. On the contrary, the range of the box plots of Figure 2 strongly increases with 3-command listeners. This means that the 3+-command data sets are more variable than for the 1- and 2-command data sets.

Figure 3 depicts the number of commits per LoC (*i.e., COMMIT*) of the analyzed GUI listeners. These results are also

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

detailed in Table 1. We observe that COMMIT does not constantly increases over CMD. This observation is assessed by the absence of correlation between these two variables (0.0570), even if this result is not statistically significant with a p-value of 0.111. We can, however, observe in Figure 3 an increase of COMMIT for the three-command listeners.

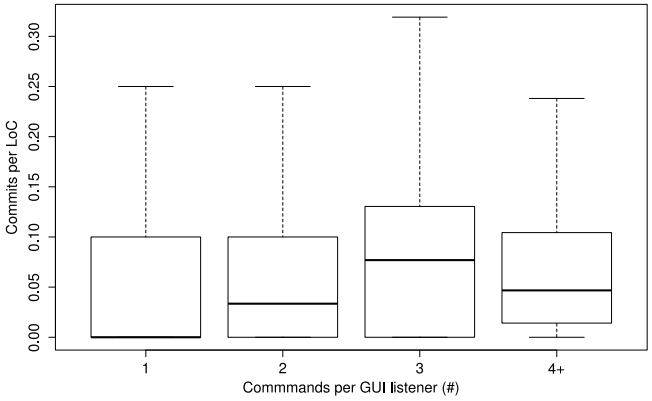

Figure 3. Number of commits per LoC of GUI listeners

Regarding **RQ2**, on the basis of these results we can conclude that *there is no evidence of a relationship between the number of GUI commands per GUI listeners and the change-proneness of the GUI listener code.*

Regarding **RQ3**, we observe a significant increase of the fault fixes per LoC for 3+-command listeners. We observe a mean of 0.004 bugs per LoC for 1- and 2-command listeners, against a mean of 0.024 bugs per LoC for 3+-command listeners, as highlighted by Figure 2. We apply the independent samples Mann-Whitney test to compare 1- and 2-command listeners against 3+-command listeners and we obtain a p-value of 2.2×10^{-16} (*i.e.,* p-value$<$0.05). We observe similar, but not significant, increase on the commits per LoC for the three-command listeners. We thus state that a threshold value, *i.e.,* a specific number of GUI commands per GUI listener, that characterizes a GUI design smell exists. On the basis of the results, *we define this threshold to three GUI commands per GUI 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 [15, 28]. Using the threshold value of 3, the concerned GUI listeners represent 21% of the analyzed GUI listener and 0.54 % of the Java code of the analyzed software systems. Besides, the average size of the 3+-command listeners is 42 LoCs, *i.e.,* less than the *long method* design smell defined between 100 and 150 LoCs in mainstream code analysis tools [1].

To conclude on this empirical study we highlight the main findings. The relation between the number of bug fixes over the number of GUI commands is too low to draw conclusions. Future works will include a larger empirical study to investigate more in depth this relation. However, a significant increase of the fault fixes per LoC for 3+-command listeners is observed. We thus set to three the number of GUI commands beyond which a GUI listener is considered as badly designed. This threshold value is an indication and as any design smell it may be defined by system experts according to the context. We show that 0.54 % of the Java code of the analyzed software systems is affected by this new GUI design smell that concerns 21% of the analyzed GUI listeners. The threats to validity of this empirical study are discussed in Section 6.1.

BLOB LISTENER: DEFINITION & ILLUSTRATION

This section introduces the GUI design smell, we call *Blob listener*, identified in the previous section, and illustrates it through real examples.

Blob Listener

We define the *Blob listener* as follows:

Definition 2 (Blob Listener) *A* Blob listener *is a GUI listener that can produce more than two GUI commands.* Blob listeners *can produce several commands because of the multiple widgets they have to manage. In such a case,* Blob listeners' *methods (such as* actionPerformed*) may be composed of a succession of conditional statements that: 1) check whether the widget that produced the event to treat is the good one,* i.e., *the widget that responds a user interaction; 2) execute the command when the widget is identified.*

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

Comparing a property of the widget. Listing 2 is an example of the first variant of *Blob listener*: the widgets that produced the event (lines 9, 12, and 14) are identified with a *String* associated to the widget and returned by *getActionCommand* (line 8). Each of the three *if* blocks forms a GUI *command* to execute in response of on the triggered widget (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 2. Widget identification using widget's properties in Swing

In Java Swing, the properties used to identify widgets are mainly the *name* or the *action command* of these widgets. The action command is a string used to identify the kind of commands the widget will trigger. Listing 3, related to Listing 2,

shows how an action command (lines 2 and 6) and a listener (lines 3 and 7) can be associated to a widget in Java Swing during the creation of the user interface.

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

Listing 3. Initialization of Swing widgets to be controlled by the same listener

Checking the type of the widget. The second variant of *Blob listener* consists of checking the *type* of the widget that produced the event. Listing 4 depicts such a practice where the type of the widget is tested using the operator *instanceof* (Lines 3, 5, 7, and 9). One may note that such *if* statements may have nested *if* statements to test properties of the widget as explained in the previous point.

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

Listing 4. Widget identification using the operator *instanceof*

Comparing widget references. The last variant of *Blob listener* consists of comparing widget references to identify those at the origin of the event. Listing 5 illustrates this variant where *getSource* returns the source widget of the event that is compared to widget references contained by the listener (*e.g.,* lines 2, 4, and 6).

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

Listing 5. Comparing widget references

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

The three variants of the *Blob listener* design smell also appear in others Java GUI toolkits, namely SWT, GWT, and JavaFX. Examples for these toolkits are available on the companion webpage of this paper[2].

AUTOMATIC DETECTION OF GUI COMMANDS AND BLOB LISTENERS

Approach Overview

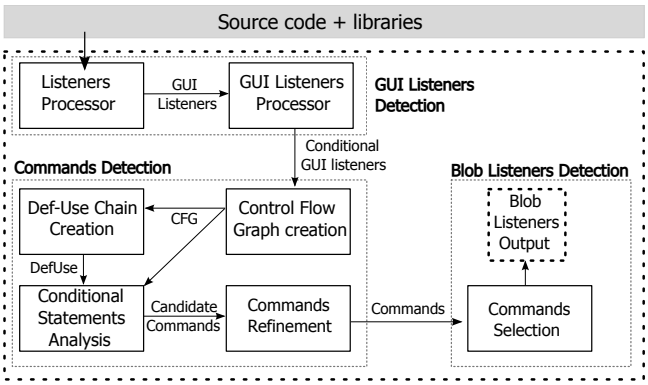

Figure 4. The proposed process for automatically detecting *Blob listeners*

Figure 4 describes the process we propose to automatically detect *Blob listeners*. The detection process includes three main steps. First, GUI listeners that contain conditional blocks (conditional GUI listeners) are automatically detected in the source code through a static analysis (Section 4.2). Then, the GUI commands, produced while interacting with widgets, that compose conditional GUI listeners are automatically detected using a second static analysis (Section 4.3). This second static analysis permits to spot the GUI listeners that are *Blob listeners*, *i.e.,* those having more than two commands. `InspectorGuidget` uses *Spoon*, a library for transforming and analyzing Java source code [29], to support the static analyses.

Detecting Conditional GUI Listeners

We define a conditional GUI listener as follows:

Definition 3 (Conditional GUI listener) *A conditional GUI listener is a listener composed of conditional blocks used to identify the widget that produced an event to process. Such conditional blocks may encapsulate a command to execute in reaction to the event.*

For instance, five nested conditional blocks (Lines 7, 9, 10, 12, and 14) compose the listener method *actionPerformed* in Listing 2 (Section 3). The first conditional block checks the type of the widget that produced the event (Line 7). This block contains three other conditional blocks that identify the widget using its action command (Lines 9, 12, and 14). Each of these three blocks encapsulates one command to execute in reaction of the event.

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

those statements are considered as conditional GUI listeners (Line 10).

Algorithm 1 Conditional GUI Listeners Detection

Input: *classes*, the source classes of the software system
Input: *tkClasses*, the classes of the GUI toolkit
Output: *listeners*, the detected conditional GUI listeners

```
1:  GUIListeners ← ∅
2:  listeners ← ∅
3:  foreach c ∈ classes do
4:      foreach tkc ∈ tkClasses do
5:          if c.isSubtypeOf(tkc) then
6:              GUIListeners ← GUIListeners ∪ getMethods(c,tkc)
7:  foreach listener ∈ GUIListeners do
8:      statements ← getStatements(listener)
9:      if hasConditional(statements) then
10:         listeners ← listeners ∪ {listener}
```

Detecting Commands in Conditional GUI Listeners

Algorithm 2 details the detection of GUI commands. The input is a set of GUI conditional listeners. The statements of conditional GUI listeners are processed to detect commands. First, we build the control-flow graph (CFG) of each listener (Line 6). Second, we traverse the CFG to gather all the conditional statements that compose a given statement (Line 7). Next, these conditional statements are analyzed to detect any reference to a GUI event or widget (Line 8). Typical references we found are for instance:

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

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

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

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

A conditional block statement can be surrounded by other conditional blocks. Potential commands detected in the function *getPotentialCmds* can thus be nested within other commands. We define such commands as *nested commands*. In such a case, the algorithm analyzes the nested conditional blocks to detect the most representative command. We observed two cases: 1. A potential command contains only a single potential command, recursively. The following code excerpt depicts this case. Two potential commands compose this code. Command #1 (Lines 1–5) has a set of statements (*i.e.,* command #2) to be executed when the widget labeled *"Copy"* is *pressed*. However, command #2 (Lines 2–4) only checks whether there is a text typed into the widget *"output"* to then allow the execution of command #1. So, command #2 works as a precondition to command #1, which is the command executed in reaction to that

Algorithm 2 Commands Detection

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

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

interaction. In this case, only the first one will be considered as a GUI command.

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

2. A potential command contains more than one potential command. The following code excerpt depicts this case. Four potential commands compose this code (Lines 1, 3, 5, and 7). In this case, the potential commands that contain multiple commands are not considered. In our example, the first potential command (Line 1) is ignored. One may note that this command checks the type of the widget, which is a variant of *Blob listener* (see Section 3.1). The three nested commands, however, are the real commands triggered on user interactions.

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

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

as *Blob listeners*. `InspectorGuidget` allows the setting of this threshold value to let system experts the possibility to adjust them, as suggested by several studies [15, 28].

EVALUATION

To evaluate the efficiency of our detection algorithm, we address the two following research questions:

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

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

The evaluation has been conducted using `InspectorGuidget`, our implementation of the *Blob listener* detection algorithm. `InspectorGuidget` is an Eclipse plug-in that analyzes Java Swing software systems. `InspectorGuidget` leverages the Eclipse development environment to raise warnings in the Eclipse Java editor on detected Blob listeners and their GUI commands. Initial tests have been conducted on software systems not reused in this evaluation. `InspectorGuidget` and all the material of the evaluation are freely available on the companion web page[2].

Objects

We conducted our evaluation by selecting six well-known or large open-source software systems based on the Java Swing toolkit: *FastPhotoTagger*, *GanttProject*, *JAxoDraw*, *Jmol*, *TerpPaint*, and *TripleA*. We use other software systems than those used in our empirical study (Section 2) to diversify the data set used in this work and assess the validation of the detection algorithm on other systems. Only *GanttProject* is part of both experiments since it is traditionally used in experiments on design smells. Table 2 lists these systems and some of their characteristics such as their number of GUI listeners.

Methodology

The accuracy of the static analyses that compose the detection algorithm is measured by the *recall* and *precision* metrics [24]. We ran `InspectorGuidget` on each software system to detect GUI listeners, commands, and *Blob listeners*. We assume as a precondition that only GUI listeners are correctly identified by our tool. Thus, to measure the precision and recall of our automated approach, we manually analyzed all the GUI listeners detected by `InspectorGuidget` to:

- *Check conditional GUI Listeners*. For each GUI listener, we manually checked whether it contains at least one conditional GUI statement. The goal is to answer RQ4 and RQ5 more precisely, by verifying whether all the conditional GUI listeners are statically analyzed to detect commands and *Blob listeners*.
- *Check commands*. We analyzed the conditional statements of GUI listeners to check whether they encompass commands. Then, *recall* measures the percentage of relevant commands that are detected (Equation (1)). *Precision* measures the percentage of detected commands that are relevant (Equation (2)).

$$Recall_{cmd}(\%) = \frac{|\{RelevantCmds\} \cap \{DetectedCmds\}|}{|\{RelevantCmds\}|} \times 100 \quad (1)$$

$$Precision_{cmd}(\%) = \frac{|\{RelevantCmds\} \cap \{DetectedCmds\}|}{|\{DetectedCmds\}|} \times 100 \quad (2)$$

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

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

$$Recall_{blob}(\%) = \frac{|\{RelevantBlobs\} \cap \{DetectedBlobs\}|}{|\{RelevantBlobs\}|} \times 100 \quad (3)$$

$$Precision_{blob}(\%) = \frac{|\{RelevantBlobs\} \cap \{DetectedBlobs\}|}{|\{DetectedBlobs\}|} \times 100 \quad (4)$$

Relevant *Blob listeners* are all the GUI listeners that handle more than two commands (see Section 4). Detecting *Blob listeners* is therefore dependent on the commands detection accuracy.

Results and Analysis

RQ4: Command Detection Accuracy. Table 3 shows the number of commands successfully detected per software system. *TripleA* has presented the highest number of GUI listeners (559), conditional GUI listeners (174), and commands (152). One can notice that despite the low number of conditional GUI listeners that has *TerpPaint* (4), this software system has 34 detected commands. So, according to the sample we studied, the number of commands does not seem to be correlated to the number of conditional GUI listeners.

Table 3. Command Detection Results

Software System	Successfully Detected Commands (#)	FN (#)	FP (#)	$Recall_{cmd}$ (%)	$Precision_{cmd}$ (%)
FastPhotoTagger	30	4	0	88.24	100.00
GanttProject	19	6	0	76.00	100.00
JaxoDraw	99	3	2	97.06	98.02
Jmol	103	18	2	85.12	98.10
TerpPaint	34	1	0	97.14	100.00
TripleA	152	44	0	77.55	100.00
OVERALL	437	76	4	85.89	99.10

Table 3 also reports the number of FN and FP commands, and the values of the *recall* and *precision* metrics. *TripleA* and *Jmol* revealed the highest number of FN, whereas *TerpPaint* presented the lowest number of FN. The *precision* of the command detection is 99.10 %. Most of the commands (437/441) detected by our algorithm are relevant. We, however, noticed 76 relevant commands not detected leading to an average *recall* of 85.89 %. Thus, our algorithm is less accurate in detecting all the commands than in detecting the relevant ones. For example, *TripleA* revealed 44 FN commands and no

Table 2. Selected interactive systems and some of their characteristics

Software Systems	Version	GUI Listeners (# (LoC))	Conditional GUI Listeners (# (LoC))	Source Repository Link
FastPhotoTagger	2.3	94 (555)	23 (408)	http://sourceforge.net/projects/fastphototagger/
GanttProject	2.0.10	67 (432)	14 (282)	https://code.google.com/p/ganttproject/
JaxoDraw	2.1	123 (1331)	50 (1128)	http://jaxodraw.svn.sourceforge.net/svnroot/jaxodraw/trunk/jaxodraw
Jmol	14.1.13	248 (1668)	53 (1204)	http://svn.code.sf.net/p/jmol/code/trunk/Jmol
TerpPaint	3.4	272 (1089)	4 (548)	http://sourceforge.net/projects/terppaint/files/terppaint/3.4/
TripleA	1.8.0.3	559 (6138)	174 (4321)	https://svn.code.sf.net/p/triplea/code/trunk/triplea/

false positive result, leading to a recall of 77.55 % and a precision of 100 %. The four FP commands has been observed in *JAxoDraw* (2) and *Jmol* (2), leading to a precision of 98.02 % and 98.10 % respectively.

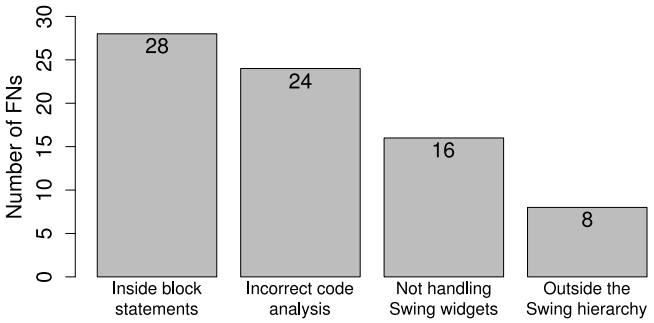

Figure 5. Distribution of the false negative commands

Figure 5 classifies the 76 FN commands according to the cause of their non-detection. 28 commands were not detected because of the use of widgets *inside block statements* rather than inside the conditional statements. For example, their conditional expressions refer to boolean or integer types rather than widget or event types. 16 other commands were not detected since they rely on *ad hoc* widgets or GUI listeners. These widgets are developed for a specific purpose and rely on specific user interactions and complex data representation [19]. Thus, our approach cannot identify widgets that are not developed under Java Swing toolkit. All the FN commands reported in this category concern *TripleA* (14) and *Jmol* (2) that use several *ad hoc* widgets. Similarly, we found eight FN commands that use classes defined *outside the Swing class hierarchy*. A typical example is the use of widgets' models (*e.g.,* classes *ButtonModel* or *TableModel*) in GUI listeners. Also, we identified 24 FN commands caused by an *incorrect code analysis* (either bugs in `InspectorGuidget` or in the *Spoon* library). This result was mainly affected by *Jmol*, that has a listener with 14 commands not detected.

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

RQ5: Blob Listeners Detection Accuracy. Table 4 gives an overview of the results of the *Blob listeners* detection per software system. The highest numbers of detected *Blob listeners* concern *TripleA* (11), *Jmol* (11), and *JAxoDraw* (7). Only one false positive and false negative have been identified against 37 *Blob listeners* correctly detected. The average recall is 97.59 % and the average precision is 97.37 %. The average time spent to analyze the software systems is 10810 ms. It includes the time that *Spoon* takes to process all classes plus the time to detect GUI commands and *Blob listeners*. The worst-case is measured in *TripleA*, *i.e.,* the largest system, with 16732 ms. *Spoon* takes a significant time to load the classes for large software systems (*e.g.,* 12437 ms out of 16732 ms in *TripleA*). Similarly to the command detection, we did not observe a correlation between the number of conditional GUI listeners, commands, and *Blob listeners*. So, regarding the recall and the precision, our approach is efficient for detecting *Blob listeners*.

Table 4. Blob Listener Detection Results

Software System	Successfully Detected *Blob listeners* (#)	FN (#)	FP (#)	$Recall_{blob}$ (%)	$Precision_{blob}$ (%)	Time (ms)
FastPhotoTagger	3	0	0	100.00	100.00	3445
GanttProject	2	0	0	100.00	100.00	1910
JAxoDraw	7	0	1	100.00	87.50	13143
Jmol	11	1	0	91.67	100.00	16904
TerpPaint	3	0	0	100.00	100.00	12723
TripleA	11	0	0	100.00	100.00	16732
OVERALL	37	1	1	97.59	97.37	10810

Regarding the single FN *Blob listener*, located in the *Jmol* software system, this FN is due to an error in our implementation. Because of a problem in the analysis of variables in the code, 14 GUI commands were not detected. Listing 6 gives an example of the FP *Blob listener* detected in *JAxoDraw*. It is composed of three commands based on checking the *states of widgets*. For instance, the three commands rely on the selection of a list (Lines 4, 7, and 11).

```
1  public final void valueChanged(ListSelectionEvent e) {
2    if (!e.getValueIsAdjusting()) {
3      final int index = list.getSelectedIndex();
4      if (index == -1) {//Command #1
5        removeButton.setEnabled(false);
6        packageName.setText("");
7      } else if ((index == 0) || (index == 1) {//Command #2
8          || (index == 2))
9          removeButton.setEnabled(false);
10         packageName.setText("");
11     } else {//Command #3
12       removeButton.setEnabled(true);
13       String name = list.getSelectedValue().toString();
14       packageName.setText(name);
15     }
16 }}
```

Listing 6. GUI code excerpt, from *JAxoDraw*

DISCUSSION

In the next three subsections, we discuss the threats to validity of the experiments detailed in this paper, the scope of `InspectorGuidget`, and alternative coding practices that can be used to limit *Blob listeners*.

Threats to validity

External validity. This threat concerns the possibility to generalize our findings. We designed the experiments using multiple Java Swing open-source software systems to diversify the observations. These unrelated software systems are developed by different persons and cover various user interactions. Several selected software systems have been used in previous research work, *e.g., GanttProject* [10, 3], *Jmol* [3], and *TerpPaint* [9] that have been extensively used against GUI testing tools. Our implementation and our empirical study (Section 2) focus on the Java Swing toolkit only. We focus on the Java Swing toolkit because of its popularity and the large quantity of Java Swing legacy code. We provide on the companion web page examples of *Blob listeners* in other Java GUI toolkits, namely GWT, SWT, and JavaFX[2].

Construct validity. This threat relates to the perceived overall validity of the experiments. Regarding the empirical study (Section 2), we used `InspectorGuidget` to find GUI commands in the code. `InspectorGuidget` might not have detected all the GUI commands. We show in the validation of this tool (Section 5) that its precision (99.10) and recall (86.05) limit this threat. Regarding the validation of `InspectorGuidget`, the detection of FNs and FPs have required a manual analysis of all the GUI listeners of the software systems. To limit errors during this manual analysis, we added a debugging feature in `InspectorGuidget` for highlighting GUI listeners in the code. We used this feature to browse all the GUI listeners and identify their commands to state whether these listeners are *Blob listeners*. During our manual analysis, we did not notice any error in the GUI listener detection. We also manually determined whether a listener is a *Blob listener*. To reduce this threat, we carefully inspected each GUI command highlighted by our tool.

Scope of the Approach

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

Second, our solution is limited to analyze GUI listeners and associated class attributes. We identified several GUI listeners that dispatch the event processing to methods. Our implemented static analyses can be extended to traverse these methods to improve its performance.

Last, the criteria for the *Blob listeners* detection should be augmented by inferring the related commands. For example, when a GUI listener is a *Blob listener* candidate, our algorithm should analyze its commands by comparing their commonalities (*e.g.,* shared widgets and methods). The goal is to detect commands that form in fact a single command.

Alternative Practices

We scrutinized GUI listeners that are not *Blob listeners* to identify alternative practices that may limit *Blob listeners*. In most of the cases, these practices consist of producing one command *per* listener by managing *one* widget per listener.

Listeners as anonymous classes. Listing 7 is an example of this good practice. A listener, defined as an anonymous class (Lines 3–7), registers with one widget (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 widget, *if* statements used to identify the involved widget are not more used, simplifying the code.

```
1  private void registerWidgetHandlers() {
2   view.resetPageButton().addActionListener(
3     new ActionListener() {
4     @Override public void actionPerformed(ActionEvent e) {
5       requestData(pageSize, null);
6     }
7   });
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 7. Good practice for defining controllers: one widget per listener

Listeners as lambdas. Listing 8 illustrates the same code than Listing 7 but using Lambdas supported since Java 8. Lambdas simplify the implementation of anonymous class that have a single method to implement.

```
1  private void registerWidgetHandlers() {
2   view.resetPageButton().addActionListener(
3                e -> requestData(pageSize, null));
4
5   view.previousPageButton().addActionListener(e -> {
6    if (hasPreviousBookmark())
7      requestData(pageSize, getPreviousBookmark());
8   });
9
10  //...
11 }
```

Listing 8. Same code than in Listing 7 but using Java 8 Lambdas

Listeners as classes. In some cases, listeners have to manage different intertwined methods. This case notably appears when developers want to combine several listeners or methods of a single listener to develop a more complex user interaction. For example, Listing 9 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  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 9. A GUI listener defined as a class

RELATED WORK

Work related to this paper fall into two categories: design smell detection; GUI maintenance and evolution.

Design Smell Detection

The characterization and detection of object-oriented (OO) design smells have been widely studied [32]. For instance, research works characterized various OO design smells associated with code refactoring operations [12, 8]. 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 [39], understandability [2], and change- or fault-proneness [16]. While developing seminal advances on OO design smells, these research works focus on OO concerns only. Improving the validation and maintenance of GUI code implies a research focus on GUI design smells, as we propose in this paper.

Related to GUI code analysis, Silva *et al.* propose an approach to inspect GUI source code as a reverse engineering process [36, 34]. Their goal is to provide developers with a framework supporting the development of GUI metrics and code analyzes. They also applied standard OO code metrics on GUI code [35]. Closely, Almeida *et al.* propose a first set of usability smells [4]. These works do not focus on GUI design smell and empirical evidences about their existence, unlike the work presented in this paper.

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 [27]. 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 [21]. Search-based techniques are also used to exploit OO code metrics [33], as well as machine learning [40], or bayesian networks [17]. Still, these works do not cover GUI design smells. In this paper, we focus on static code analysis to detect GUI commands to form a *Blob listener* detection rule. To do so, we use a Java source code analysis framework that permits the creation of specific code analyzers [29]. Future work may investigate other heuristics and analyses to detect GUI 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 [22]. Garcia *et al.* propose an approach for identifying architectural design smells [14]. Similarly, this work aims at motivating that GUIs 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 [25, 11] or how they are perceived by developers [28]. Future work may focus on how software developers perceive *Blob listeners*.

GUI maintenance and evolution

Unlike object-oriented design smells, less research work focuses on GUI design smells. Zhang *et al.* propose a technique to automatically repair broken workflows in Swing GUIs [42]. Static analyses are proposed. This work highlights the difficulty "*for a static analysis to distinguish UI actions [GUI commands] that share the same event handler [GUI listener]*". In our work, we propose an approach to accurately detect GUI commands that compose GUI listeners. Staiger also proposes a static analysis to extract GUI code, widgets, and their hierarchies in C/C++ software systems [38]. The approach, however, is limited to find relationships between GUI elements and thus does not analyze GUI controllers and their listeners. Zhang *et al.* propose a static analysis to find violations in GUIs [41]. These violations occur when GUI operations are invoked by non-UI threads leading a GUI error. The static analysis is applied to infer a static call graph and check the violations. Frolin *et al.* propose an approach to automatically find inconsistencies in MVC JavaScript applications [24]. GUI controllers are statically analyzed to identify consistency issues (*e.g.,* inconsistencies between variables and controller functions). This work is highly motivated by the weakly-typed nature of Javascript.

CONCLUSION

In this paper, we investigate a new research area on GUI design smells. We detail a specific GUI design smell, we call *Blob listener*, that can affect GUI listeners. The empirical study we conducted exhibits a specific number of GUI commands per GUI listener that characterizes a *Blob listener* exists. We define this threshold to three GUI commands per GUI listener. We show that 21 % of the analyzed GUI controllers are affected by *Blob listeners*. We propose an algorithm to automatically detect *Blob listeners*. This algorithm has been implemented in a tool publicly available and then evaluated.

Next steps of this work include a behavior-preserving algorithm to refactor detected *Blob listeners*. We will conduct a larger empirical study to investigate more in depth the relation between the number of bug fixes over the number of GUI commands. We will study different GUI coding practices to identify other GUI design smells. We will investigate whether some GUI faults [19] are accentuated by GUI design smells.

Acknowledgements

This work is partially supported by the French BGLE Project CONNEXION. We thank Yann-Gaël Guéhéneuc for his insightful comments on this paper.

REFERENCES

1. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11, 2 (2012), 1–38. DOI: http://dx.doi.org/10.5381/jot.2012.11.2.a5
2. Marwen Abbes, Foutse Khomh, Y. G. Guéhéneuc, and Giuliano Antoniol. 2011. 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*. 181–190. DOI: http://dx.doi.org/10.1109/CSMR.2011.24
3. Andrea Adamoli, Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. 2011. Automated GUI performance testing. *Software Quality Journal* 19, 4 (2011), 801–839. http://dx.doi.org/10.1007/s11219-011-9135-x
4. Diogo Almeida, José Creissac Campos, João Saraiva, and João Carlos Silva. 2015. Towards a Catalog of Usability Smells. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. ACM, 175–181. http://doi.acm.org/10.1145/2695664.2695670
5. Michel Beaudouin-Lafon. 2000. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. of CHI'00*. ACM, 446–453. http://dl.acm.org/citation.cfm?id=332473
6. Arnaud Blouin and Olivier Beaudoux. 2010. Improving modularity and usability of interactive systems with Malai. In *Proc. of EICS'10*. 115–124. https://hal.inria.fr/inria-00477627
7. Arnaud Blouin, Brice Morin, Grégory Nain, Olivier Beaudoux, Patrick Albers, and Jean-Marc Jézéquel. 2011. 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*. 85–94. DOI: http://dx.doi.org/10.1145/1996461.1996500
8. William J Brown, Hays W McCormick, Thomas J Mowbray, and Raphael C Malveau. 1998. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley New York.
9. M.B. Cohen, Si Huang, and A.M. Memon. 2012. AutoInSpec: Using Missing Test Coverage to Improve Specifications in GUIs. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. 251–260. http://dx.doi.org/10.1109/ISSRE.2012.33
10. Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11, 2 (2012), 5:1–38. http://www.jot.fm/issues/issue_2012_08/article5.pdf
11. Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bartosz Walter, and Pawel Martenka. 2013. 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, 260–269. http://dx.doi.org/10.1109/ICSM.2013.37
12. Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley.
14. Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. 2009. Identifying architectural bad smells. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 255–258. http://dx.doi.org/10.1109/CSMR.2009.59
15. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? *Proc. of International Conference on Software Engineering* (2013), 672–681. DOI: http://dx.doi.org/10.1109/ICSE.2013.6606613
16. Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering* 17, 3 (2012), 243–275. DOI: http://dx.doi.org/10.1007/s10664-011-9171-y
17. Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572. http://www.sciencedirect.com/science/article/pii/S0164121210003225
18. G. E. Krasner and S. T. Pope. 1988. A Description of the Model-View-Controller User Interface Paradigm in Smalltalk80 System. *Journal of Object Oriented Programming* 1 (1988), 26–49.
19. Valéria Lelli, Arnaud Blouin, and Benoit Baudry. 2015. Classifying and Qualifying GUI Defects. In *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*. IEEE. https://hal.inria.fr/hal-01114724v1
20. Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. 2007. Assessing the impact of bad smells using historical information. In *Workshop on Principles of software evolution*. ACM, 31–34. http://dx.doi.org/10.1145/1294948.1294957
21. Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and A Le Meur. 2010. DECOR: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on* 36, 1 (2010), 20–36. http://dx.doi.org/10.1109/TSE.2009.50

22. Naouel Moha, Francis Palma, Mathieu Nayrolles, Benjamin Joyen Conseil, Guéhéneuc Yann-Gael, Benoit Baudry, and Jean-Marc Jézéquel. 2012. Specification and Detection of SOA Antipatterns. In *International Conference on Service Oriented Computing*. **https://hal.inria.fr/hal-00722472**

23. Brad A. Myers. 1991. 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)*. ACM, 211–220. **http://dl.acm.org/citation.cfm?id=120805**

24. Frolin Ocariza, Karthik Pattabiraman, and Ali Mesbah. 2015. Detecting Inconsistencies in JavaScript MVC Applications. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 11 pages. **https://dl.acm.org/citation.cfm?id=2818796**

25. Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjoberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Proc. of ICSM'10*. IEEE, 1–10. **http://dx.doi.org/10.1109/ICSM.2010.5609564**

26. Jason W. Osborne and Averil Overton. 2004. The power of outliers (and why researchers should always check for them). *Practical Assessment, Research & Evaluation* 9, October (2004). **http://pareonline.net/getvn.asp?v=9&n=6+**

27. F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2014a. Mining Version Histories for Detecting Code Smells. *Software Engineering, IEEE Transactions on* (2014). **http://dx.doi.org/10.1109/TSE.2014.2372760**

28. Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014b. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *Proc. of ICSM'14*. IEEE, 101–110. **http://dx.doi.org/10.1109/ICSME.2014.32**

29. Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. SPOON: A library for implementing analyses and transformations of Java source code. *Software: Practice and Experience* 43, 4 (2015). DOI:**http://dx.doi.org/10.1002/spe.2346**

30. Mike Potel. 1996. MVP: Model-View-Presenter the Taligent Programming Model for C++ and Java. *Taligent Inc* (1996).

31. D Rapu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. 2004. Using history information to improve design flaws detection. In *Proc. of Conference on Software Maintenance and Reengineering*. 223–232. **http://dx.doi.org/10.1109/CSMR.2004.1281423**

32. Ghulam Rasool and Zeeshan Arshad. 2015. A review of code smell mining techniques. *Journal of Software: Evolution and Process* 27, 11 (2015), 867–895.

33. Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. 2014. Code-Smell Detection As a Bilevel Problem. *ACM Trans. Softw. Eng. Methodol.* 24, 1, Article 6 (Oct. 2014), 44 pages. DOI: **http://dx.doi.org/10.1145/2675067**

34. João Carlos Silva, Carlos Silva, Rui D. Gonçalo, João Saraiva, and José Creissac Campos. 2010. The GUISurfer Tool: Towards a Language Independent Approach to Reverse Engineering GUI Code. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*. ACM, 181–186. **http://doi.acm.org/10.1145/1822018.1822045**

35. João Carlos Silva, J Creissac Campos, J Saraiva, and José L Silva. 2014. An approach for graphical user interface external bad smells detection. In *New Perspectives in Information Systems and Technologies*. 199–205. DOI: **http://dx.doi.org/10.1007/978-3-319-05948-8_19**

36. Joao Carlos Silva, J Creissac Campos, and João Alexandre Saraiva. 2010. GUI inspection from source code analysis. (2010). **http://hdl.handle.net/1822/18517**

37. Josh Smith. 2009. WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine* (February 2009). **http://msdn.microsoft.com/en-us/magazine/dd419663.aspx**

38. S. Staiger. 2007. Static Analysis of Programs with Graphical User Interface. In *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*. 252–264. **http://dx.doi.org/10.1109/CSMR.2007.44**

39. Aiko Yamashita and Leon Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: an empirical study. In *35th International Conference on Software Engineering, ICSE '13*. 682–691. DOI: **http://dx.doi.org/10.1109/ICSE.2013.6606614**

40. Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. 2015. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software* 103, 0 (2015), 102 – 117. DOI: **http://dx.doi.org/10.1016/j.jss.2015.01.037**

41. Sai Zhang, Hao Lü, and Michael D. Ernst. 2012. Finding Errors in Multithreaded GUI Applications. In *Proc. ISSTA'12 (ISSTA 2012)*. ACM, 243–253. **http://doi.acm.org/10.1145/2338965.2336782**

42. Sai Zhang, Hao Lü, and Michael D. Ernst. 2013. Automatically repairing broken workflows for evolving GUI applications. In *ISSTA 2013, Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 45–55. **http://doi.acm.org/10.1145/2483760.2483775**