And the Tool Created a GUI That was Impure and Without Form: Anti-Patterns in Automatically Generated GUIs

Apostolos V. Zarras
Department of Computer Science and Engineering, University of Ioannina, Greece
zarras@cs.uoi.gr

Georgios Mamalis
Department of Computer Science and Engineering, University of Ioannina, Greece
gmamalis@cs.uoi.gr

Aggelos Papamichail
Department of Computer Science and Engineering, University of Ioannina, Greece
apapamichail@cs.uoi.gr

Panagiotis Kollias
Department of Computer Science and Engineering, University of Ioannina, Greece
cs101862@cs.uoi.gr

Panos Vassiliadis
Department of Computer Science and Engineering, University of Ioannina, Greece
pvassil@cs.uoi.gr

ABSTRACT

A basic prerequisite for any daily development task is to understand the source code that we are working with. To this end, the source code should be clean. Usually, it is up to us, the developers, to keep the source code clean. However, often there are parts of the code that are automatically generated. A typical such case are Graphical User Interfaces (GUIs) created via a GUI builder, i.e., a tool that allows the developer to design the GUI by combining graphical control elements, offered in a palette. In this paper, we investigate the quality of the code that is generated by GUI builders. To assist tool-smiths in developing better GUI builders, we report anti-patterns concerning naming, documentation, design and implementation issues, observed in a study that involves four popular GUI builders for Java. The reported anti-patterns can further assist GUI developers/designers in selecting appropriate tools.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**;

KEYWORDS

GUIs, Patterns, Refactoring, Code Clones, Responsibilities

ACM Reference Format:
Apostolos V. Zarras, Georgios Mamalis, Aggelos Papamichail, Panagiotis Kollias, and Panos Vassiliadis. 2018. And the Tool Created a GUI That was Impure and Without Form: Anti-Patterns in Automatically Generated GUIs. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3282308.3282333

EuroPLoP '18, July 4–8, 2018, Irsee, Germany

ACM ISBN 978-1-4503-6387-7/18/07…$15.00
https://doi.org/10.1145/3282308.3282333

1 INTRODUCTION

> *"I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them.* ***Clean code*** *always looks like it was* ***written by someone who cares****."*

The previous quote, is a definition of the term *clean code* given by Michael Feathers in Robert Martin's homonymous book [15].

> *But what if that someone does not exist? What if the code is actually generated by a tool?*

In this paper, we focus on tools that generate Graphical User Interfaces (GUIs). A GUI builder allows the developers to design a GUI by combining graphical control elements, offered in a palette. Based on the GUI design, the tool generates a corresponding implementation. Typically, the developers use GUI builders for fast prototyping and testing. Following, they often restructure/refactor the GUIs that they developed with these tools, to improve the source code quality and introduce new features that enhance the static layouts supported by the tools[1]. On our side, we empirically observed that the source code of generated GUIs is typically long and complex. This was a real problem every time we had to understand how the code works, so as to link it with the underlying business logic, test it, or extend it to realize more advanced interactions that are not supported by the tool. The aforementioned issues prompted us to investigate in more detail the code that is generated by GUI builders.

So far, several empirical studies have been performed to assess the source code adherence to coding conventions and best practices (e.g., [1, 3, 4, 6, 7, 14, 18, 21]). The state of the art further comprises interesting tools and techniques that allow the developers to deal with naming (e.g., [2, 12]), documentation (e.g., [22]), design and implementation issues [9, 16]. Nevertheless, the quality of automatically generated GUIs has not been studied before. We address this shortcoming in a study that concerns four popular GUI builders for Java. Using these tools, we designed, as case studies, (partial) replicas of the Eclipse GUI. We focus on naming, documentation, design and implementation issues. Concerning these issues, we

[1]For instance, see a related forum discussion at www.reddit.com/r/java/comments/30gz4s/gui_builders_goodbad/

observed good and bad practices in the code of automatically generated GUIs. We believe that reporting both the good and the bad practices involved in the context of a particular issue would be useful. For this reason, we choose to report our findings in the form of anti-patterns. Typically, an anti-pattern specifies a problematic solution to a specific problem, along with a better solution to the problem [5, 13]. The reported anti-patterns are primarily intended to help tool-smiths in developing better GUI builders, and secondly to assist GUI developers/designers in the tool selection process.

The rest of the paper is structured as follows. Section 2, provides the necessary background concerning our empirical study. Sections 3, 4, 5, and 6 introduce the observed anti-patterns. Section 7 discusses related (anti-)patterns. Finally, Section 8 summarizes our contribution.

2 EMPIRICAL STUDY SETUP

In this section, we detail the setup of our study. Moreover, we discuss threats to validity, concerning the study and the observed anti-patterns. Finally, we introduce the template that we employ for the specification of the anti-patterns.

GUI Builders. In our study, we considered four popular GUI builders for Java: *Matisse*[2], *JFormDesigner*[3], *WindowBuilder*[4], and *SceneBuilder*[5]. All of the tools provide a palette that allows the developer to select, via drag and drop, the different elements (a.k.a. widgets) of a GUI. Typically, the GUI widgets are *top-containers* (e.g., windows, frames, applets), *lightweight containers* (e.g., dialogs, panels), *control elements* (e.g., buttons, checkboxes, text fields, text areas), and *menu elements* (e.g., menu bars, menus, menu items). Using the tools, we designed, as case studies, partial replicas of the Eclipse GUI. Each replica realizes the main perspective of the Eclipse GUI, which includes the IDE's menu structure, toolbar, editor, console and package explorer. With Matisse and JFormDesigner we developed Swing[6] case studies; with WindowBuilder we constructed two case studies that rely on Swing and SWT[7]; with SceneBuilder we developed a JavaFX[8] case study. The analysis of the generated GUIs and the respective anti-patterns that are detailed in the next sections focus on the following key issues of source code quality [15]: the *names* of the generated source code elements; the *comments* that document the generated source code; the *design/implementation* of classes and methods. To analyze the code of the generated GUIs we used CheckStyle [8]. We configured CheckStyle according to the Oracle standard coding style for Java [9]. To check whether the names of classes, methods and variables consist of terms that belong to the right part of speech (e.g., method names beginning with verbs) we extended the tool via WordNet[10], a lexical database developed at the Princeton University.

Threats to Validity. To ensure the *construct validity* of the study, we used CheckStyle [8], a well-known open-source tool, for the analysis of the generated GUIs. Moreover, the metrics that we employed for the assessment are directly related to the measured constructs. *Internal validity* is not an issue in our work as we do not attempt to establish any particular cause-effect relationship. Concerning *external validity*, we observed each one of the anti-patterns several times in multiple case studies. However, our study is focused on GUI builders for Java and related technologies. Extending the study in more case studies that rely on other tools, languages, and technologies, may result in further related anti-patterns.

Anti-patterns Template. In the literature there are several templates for the specification of patterns [11, 17, 24] and anti-patterns [5, 13]. The template that we use in this paper is inspired from the aforementioned efforts. Nevertheless, the employed template is adapted to the specificities of our work, which concerns empirically observed anti-patterns in automatically generated GUIs. Specifically, we specify an observed anti-pattern in terms of the following elements: a characteristic *name* that highlights a problematic (anti) solution, observed in automatically generated GUIs, a description of the *problem* that is solved by the problematic solution, the *anti-solution*; a brief discussion concerning *unbalanced forces* that relate with the observed anti-solution, a *better solution*, a brief report regarding *empirical evidence* of the observed anti-pattern.

3 NUMBERED WIDGETS

Problem. A typical GUI builder starts from the design of a GUI, specified using the tool's palette, and generates the GUI implementation that relies on a particular widget API. In particular, the tool maps the widgets that constitute the GUI into corresponding implementation-specific elements (classes, objects, etc.). To this end, the tool must generate meaningful names for the mapped widgets i.e., names that reveal the purpose of the widgets in the GUI.

Anti-Solution: Numbered Widgets. The GUI builder generates a widget name is generated as a combination of a prefix, derived from the type of the widget, and a serial number that allows to distinguish between widgets of the same type.

Forces. When used properly, names are a powerful tool that allows the developers to express their intention in the code that they develop. On the contrary, inappropriate names can become a curse for anyone who's trying to understand, test, or maintain a piece of code. In our context, the naming approach that combines type prefixes and serial numbers is straightforward and quite convenient for the code generation process. The widget names do not depend on any other widget property (e.g., the widget labels). Consequently, the names are generated once and for all. There is no need to maintain any kind of consistency between the generated names and other widget properties that could change overtime. However, the combination of type prefixes with serial numbers does not enable the generation of intention revealing and informative names [15]. At the same time, the use of type prefixes is not search-friendly, as it increases the similarity between widgets of the same type. Moreover, the use of serial numbers does not favor the generation of pronounceable names that make meaningful distinctions.

[2] netbeans.org/features/java/swing.html
[3] www.formdev.com/jformdesigner
[4] www.eclipse.org/windowbuilder/
[5] www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html
[6] docs.oracle.com/javase/tutorial/uiswing/start/index.html
[7] www.eclipse.org/swt/docs.php
[8] docs.oracle.com/javafx/2/overview/jfxpub-overview.htm
[9] www.oracle.com/technetwork/java/codeconventions-150003.pdf
[10] wordnet.princeton.edu

Better Solution: Named Widgets. Typically, widgets are characterized by labels, specified by the GUI designer/developer. Exploiting the provided labels gives a good chance for intention revealing widget naming.

Empirical Evidence. In our empirical study we observe numbered widgets in two case studies (Matisse and JFormDesigner). Figure 1, gives specific examples that show how Matisse and JFormDesigner actually name widgets; red/italics and cyan/underline distinguish between the type prefix and the serial number, respectively. Overall, in these two case studies 91% of the widgets (i.e., number of buttons, checkboxes, radio buttons, menu bars, menus and menu items, over the total number of widget that constitute the GUI) are characterized by intention revealing labels (i.e., labels that reveal the purpose of the widgets in the GUI), which could serve for the generation of better names. The rest of them are lightweight containers (e.g. panels), popup menus and auxiliary elements (e.g. separators).

JFormDesigner

```
javax.swing.JMenu jMenu1;
```

Matisse

```
JMenu menu1;
```

Figure 1: Examples of numbered widgets.

The other three case studies (WindowBuilder/SWT, WindowBuilder/Swing and SceneBuilder) take advantage of the widgets' labels. In the WindowBuilder case studies, the name of a labeled widget consists of a prefix, derived from the type of the widget, and the provided label, while an unlabeled widget is named by following the numbered widget approach. Similarly, in the SceneBuilder case study, labeled widgets are mapped to labeled FXML elements, while unlabeled widgets are mapped into FXML elements that are characterized only by their type. Figure 2, gives examples of this approach; red/italics and cyan/underline emphasize the type prefix and the label, respectively. Overall, the percentage of widgets that come with intention revealing names in WindowBuilder/Swing, WindowBuilder/SWT, and SceneBuilder is 91%, 86% and 90%, respectively.

WindowBuilder/Swing

```
JMenu mnFile = new JMenu("File");
```

WindowBuilder/SWT

```
MenuItem mntmFile = new MenuItem(menu);
mntmFile.setText("File");
```

SceneBuilder

```
<Menu mnemonicParsing="false" text="File">
```

Figure 2: Examples of named widgets.

4 INAPPROPRIATE GUI COMMENTS

Problem. An automatically generated GUI should be documented with informative comments that facilitate the integration of the generate code with the underlying application logic.

Anti-Solution: Inappropriate GUI Comments. The GUI builder does not generate informative comments, or it generates a mixture consisting of informative comments and comments whose purpose is not clear.

Forces. Generating no comments at all can be convenient for the GUI builder, as it simplifies the overall code generation process. However, the application developers may need help, to deal with further development tasks that involve the generated source code. In particular, the developers must integrate the generated GUI with the underlying business logic. Moreover, the automatically generated GUI could be just a prototype that should be extended with further functionalities that concern more dynamic interactions, or even refactored to improve the quality of the generated code. Informative comments that facilitate such tasks would be useful. Nevertheless, mixing comments that concern the application developers with comments that serve other purposes (e.g., tool specific comments that facilitate the code generation process) could be a burden for the application developers. Redundant or misleading comments that are less informative than the code are also undesirable [15].

Better Solution: Informative GUI Comments. The GUI builder generates the following kinds of comments:

- *Legal/authorship* comments that identify the GUI designers and the tool that generated the GUI source code.
- *Javadoc* comments for public GUI elements.
- *TODO* comments that specify "hooks" where the application developers should put the "glue" code for handling the events, generated by the widgets.
- *Warnings/clarification* comments concerning GUI implementation related issues that the GUI designer/developer should be aware of.

Comments that serve other purposes should be avoided. If not possible, the purpose of such comments should be clearly specified to allow distinguishing them from comments that concern the GUI designers/developers.

Empirical Evidence. In our study, two case studies (Matisse, JFormDesigner) are bloated with *unclear comments* comments (Table 1).

Specifically, in the Matisse case study, generated event handling methods are surrounded with opening/closing brace comments (e.g., Figure 3). Surely, these comments do not reveal the purpose of the methods themselves, which is actually reflected quite well from the methods' names and the generated TODO comments, within the body of the methods. Concerning the purpose of the opening/-closing brace comments, we could make several assumptions: they could be warnings (e.g., to prevent the GUI designers/developers from changing the methods), they could be useful for the code generation process, and so on. Nevertheless, nothing of the previous is clearly stated.

The JFormDesigner case study includes comments that mark the position of (similar) long widget configuration code blocks (e.g.,

Table 1: Classification of comments found in the case studies.

Informative comments	Matisse	JFormDesigner	WindowBuilder		SceneBuilder
			Swing	SWT	
legal authorship	1	1	NO COMMENTS		
TODO	450	453			
amplification warning	2	2			
Javadoc	4	1	2	3	
Total sum	457	457	2	3	
Unclear comments	**Matisse**	**JFormDesigner**	**WindowBuilder**		**SceneBuilder**
			Swing	SWT	
position markers	2	574	NO COMMENTS		
opening closing brace	900	2			
Total sum	902	576			

Figure 3). Normally, these comments wouldn't be needed if the code was not long, complex and full of clones (more in Section 5). Moreover, there are several comments that mark variable declaration fragments. If not used for the code generation process, these comments are quite useless because the code in these cases "speaks for itself".

Opening/Closing Braces: Matisse example

```
private void jMenuItem1ActionPerformed(ActionEvent evt) {
//GEN-FIRST:event_jMenuItem1ActionPerformed
    // TODO add your handling code here:
}
//GEN-LAST:event_jMenuItem190ActionPerformed
```

Position Marker: JFormDesigner example

```
//---- menuItem315 ----
menuItem315.setText("JPanel");
menuItem315.setIcon(new
ImageIcon(getClass().getResource(…)));
menuItem315.addActionListener(
      e -> menuItem315ActionPerformed(e));
menu44.add(menuItem315);
```

Figure 3: Examples of unclear comments.

On the positive side, the aforementioned case studies also include *informative* comments (e.g. Figure 4) like legal/authoriship comments, TODO comments for the generated event handling methods, and comments that warn the GUI developer to avoid changing the code of generated methods.

In the remaining three case studies, the provided documentation is *limited*. WindowBuilder/Swing and WindowBuilder/SWT include only few Javadoc comments, while in the SceneBuilder case there are no comments at all.

5 WIDGET CONFIGURATION CLONES

Problem. Having generated the source code that maps the widgets of a particular GUI into corresponding implementation-specific

TODO: Matisse example

```
private void jMenuItem1ActionPerformed(ActionEvent evt)
{
    // TODO add your handling code here:
}
```

Amplification: JFormDesigner example

```
private void initComponents() {
    // JFormDesigner - initialization - DO NOT MODIFY
    ……
}
```

Figure 4: Examples of informative comments.

elements (classes, objects, etc.), the GUI builder must generate the source code that configures the properties (labels, colours, coordinates, etc.) of the mapped elements using the underlying widget API.

Anti-Solution: Widget Configuration Clones. For each widget, the tool generates a respective code fragment that configures the widget's properties.

Forces. Generating a widget configuration code fragment for each widget is straightforward and simplifies the code generation process, as there is no need to come up with generic widget configuration methods. However, a GUI typically comprises many widgets of the same type (panels, menus, menu items, buttons, text fields, etc.), characterized by similar properties. Consequently, the generated GUI implementation includes a lot of duplicated code that comes in the form of Type 2, and 3 clones (i.e., duplicate fragments with added or removed statements, in addition to variations in identifiers and literals [20]).

Better Solution: General Widget Configuration Interpreter Methods. For the different types of widgets involved, have respective methods that take as input a specification of widget properties, interpret the given specification, and configure the widget using the underlying widget API. The widget's properties can be specified in a declarative way, via a configuration-specific data structure (class, map, XML schema, etc.). However, certain specification means like XML can make the program more difficult to test. Moreover, IDEs may not be able to support certain refactorings (e.g., renamings), when external configuration files are involved.

Empirical Evidence. This anti-pattern appears in four out of five case studies (Matisse, JFormDesigner, WindowBuilder/Swing, WindowBuilder/SWT). Table 2 gives more details concerning the anti-pattern. In particular, the amount of duplicate code in the case studies ranges from 1898 to 2665 lines of code. The percentage of duplicate code over the total size of each study is also notable, ranging from 23.4% to 47.7%.

Figure 5 shows two clones from the WindowBuilder/Swing case study, each responsible for the configuration of a respective menu item.

The SceneBuilder case study is an indicative example of the good solution. The properties of a particular widgets' are specified in FXML, while an FXML loader interprets the specification to

Table 2: Duplicate code statistics: size of each case study, measured as the total number of physical lines of code of its classes; size of duplicate code, measured as the total number of lines of code spent in code clones; percentage of duplicate code over the size of the case study.

	Matisse	JFormDesigner	WindowBuilder		SceneBuilder
			Swing	SWT	
total size (ploc)	8123	6572	4279	5805	82
duplicate code	1898	2014	2043	2665	0
% duplicate code	23,4%	30,6%	47,7%	45,9%	0,0%

```
JMenuItem mntmClose = new JMenuItem("Close");
mntmClose.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
  }
});
mntmClose.setEnabled(false);
mnFile.add(mntmClose);
```
Clone 1

```
JMenuItem mntmCloseAll = new JMenuItem("Close All");
mntmCloseAll.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
  }
});
mntmCloseAll.setEnabled(false);
mnFile.add(mntmCloseAll);
```
Clone 2

Figure 5: Menu item configuration clones found in the WindowBuilder/Swing case study.

configure the widget. Figure 6 illustrates the specification of menu items in FXML.

```
<Menu text="File" mnemonicParsing="false">
  ........
  ........
  <MenuItem text="Close" mnemonicParsing="false" onAction="#run"/>
  <MenuItem text="Close All" mnemonicParsing="false" onAction="#run"/>
  <MenuItem text="Save" mnemonicParsing="false" onAction="#run"/>
  ........
  ........
</Menu>
```

Figure 6: Specification of menu items in the SceneBuilder case study.

Another example concerning the good solution is given in Figure 7. More specifically, the figure gives the code of `setComponentProperties()`, a generic method that can be used to remove clones like the ones given in Figure 5. In our study, we came up with this method in an attempt to refactor the code of the WindowBuilder/Swing case study. The method relies on Java reflection; it takes as input any widget, derived from the abstract JComponent class and the specification of the widget's properties. The specification is given in the form of a map that contains property name/value pairs. Based on the name of each property, `setComponentProperties()` identifies the appropriate `JComponent` setter. Following, `setComponentProperties()` invokes the setter, giving as input the respective property value.

```
void setComponentProperties(JComponent component,
                            Map<String, Object> properties){

  Set<String> keys = properties.keySet();
  Method jComponentMethods[] = component.getClass().getMethods();

  for (String methodName : keys){
    for (int i = 0; i < jComponentMethods.length; i++){
      if (methodNamesMatch(jComponentMethods[i], methodName)){
        try{
          Object requiredParams = properties.get(methodName);
          if (requiresMultipleParameters(requiredParams) &&
              paramListsMatch(jComponentMethods[i], requiredParams)) {
              jComponentMethods[i].invoke(component,
                                  ((List) requiredParams).toArray());
              break;
          } else
            if (singleParamsMatch(jComponentMethods[i], requiredParams)){
              jComponentMethods[i].invoke(component, requiredParams);
              break;
            }
        } catch (Exception e) {
           System.err.println(e.getMessage());
           System.exit(1);
        }
      }
    }
  }
  ((JComponent) properties.get("container")).add(component);
}
```

Figure 7: A generic configuration method that can be used to remove widget configuration clones in the WindowBuilder/Swing case study.

6 MIX OF WIDGET & EVENT HANDLING CODE

Problem. To be functional, the GUI widgets must come along with event handling code that binds user interaction events with the underlying business logic. To this end, the GUI builder must generate the event handling "hooks" for the binding code.

Anti-Solution: Mix of Widget & Event Handling Code. The GUI builder maps each top-level GUI container into a class that contains both the widget configuration code and the respective event handling code.

Forces. Having the the widget configuration code and the event handling code in the same top-level container class simplifies the code generation process as it reduces the number of generated classes that should be handled by the tool. Nevertheless, this approach results in god classes [10] that mix different responsibilities.

Better Solution: Widget and Event Handling Apart. Separate the widget configuration code from the event handling code via a Model View Controller pattern [19]. An interesting issue to this end is to *decide the granularity of the controllers*. Two possible options are:

- *One controller per top level container.* The risk that comes up in this case is that the controller itself can become a god class if the container comprises many constituent widgets.
- *Different controllers/handlers for the individual widgets.* The issue in this case is bloating the GUI with many classes that do little work.

Empirical Evidence. We observe this anti-pattern in four out of five case studies (Matisse, JFormDesigner, WindowBuilder/Swing,

WindowBuilder/SWT). All of these case studies comprise a huge top-level container class. The event handling code that is mixed with the widget configuration code ranges from 1359 to 3247 lines of code, i.e., 20.7% to 40.0% of the overall implementation (Table 3).

Table 3: Event handling code statistics: size of each case study, measured as the total number of physical lines of code of its classes; size of event handling code; percentage of event handling code over the size of the case study.

	Matisse	JFormDesigner	WindowBuilder		SceneBuilder
			Swing	SWT	
total size (ploc)	8123	6572	4279	5805	82
event handling code	3247	1359	1472	1960	56
% event handling code	40,0%	20,7%	34,4%	33,8%	68,3%

Figure 8 gives a specific example of the anti-pattern from the WindowBuilder/Swing case study. In particular, `EclipseGUI` is the top-level container class of this case study. This class configures all the widgets that constitute the GUI and further contains the event handling code. The `addPopup()` method, for instance, concerns the handling of popup menu events.

```
public class EclipseGUI extends JFrame {
  private JPanel contentPane;
  private JTextField txtQuickAccess;
  private final ButtonGroup buttonGroup = new ButtonGroup();
.................
.................
.................
  private static void addPopup(Component component, final JPopupMenu popup) {
      component.addMouseListener(new MouseAdapter() {
              public void mousePressed(MouseEvent e) {
                     if (e.isPopupTrigger()) {
                            showMenu(e);
                     }
              }
              public void mouseReleased(MouseEvent e) {
                     if (e.isPopupTrigger()) {
                            showMenu(e);
                     }
              }
              private void showMenu(MouseEvent e) {
                     popup.show(e.getComponent(), e.getX(), e.getY());
              }
      });
  }
}
```

Figure 8: Mixed UI and event handling in the WindowBuilder/Swing case study.

The SceneBuilder case study is an indicative example of the good solution. The GUI implementation comprises a controller class that is responsible for the handling of events. Another example concerning the good solution is given in Figure 9, where the inline handling methods that are included in `addPopup()` of Figure 8 are extracted in a separate class, named `PopupListener`, which is responsible for the handling of popup menu events.

```
public class PopUpListener extends MouseAdapter{
    private JPopupMenu popup;
    public PopUpListener(JPopupMenu popup) {
        this.popup = popup;
    }

    public void mousePressed(MouseEvent e) {
        if (e.isPopupTrigger()) {
            showMenu(e);
        }
    }
    public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger()) {
            showMenu(e);
        }
    }
    private void showMenu(MouseEvent e) {
        popup.show(e.getComponent(), e.getX(), e.getY());
    }
}
```

Figure 9: Separate event handling in the WindowBuilder/Swing case study.

7 FURTHER RELATED (ANTI-)PATTERNS & CONCEPTS

Our work concerns automatically generated GUI implementations. Hence from a broader perspective it is related with Markus Völter's patterns for program generation [23]. In this work, the author introduced a catalog of seven patterns:

- Templates & Filtering: according to this pattern, code is generated by applying certain templates to a textual model specification (e.g. a XMI specification of a UML model).
- Templates & Meta-model: the code generation process can be customized with respect to different sets of templates that correspond to respective meta-models (e.g. UML profiles).
- Frame Processing: code is generated with respect to parameterized programs, called frames.
- API-Based Generators: the idea behind this pattern is to have the code generating programs, specific to respective APIs.
- Inline Code Generation: according to this pattern, code is generated in a pre-compile/processing step of a regular non-generated program.
- Code Attributes: concerns the generation of code that is based on annotations or attributes, specified for a regular non-generated program.
- Code Weaving: this pattern introduces a code generation process that is based on combining together different code fragments that concern respective aspects (e.g. concurrency, persistence, etc.)

Regarding our study, we can characterize the examined GUI builders as API-Based Generators, as they generate code with respect to specific GUI APIs (Swing, SWT and JavaFX).

In our study, we assessed the names of the generated code elements with respect to standard Java naming conventions. In this context, Arnaoudova et al. further proposed a catalog of linguistic anti-patterns [3]. At a glance, these anti-patterns can be divided in the following two categories:

- Anti-patterns regarding methods whose implementation semantics are not consistent with the methods' prototypes.

- Anti-patterns concerning attributes/variables, whose names are not consistent with their types.

In the generated GUI implementations, we did not observe instances of the aforementioned linguistic anti-patterns.

8 TAKEAWAY & FUTURE PERSPECTIVES

Do GUI builders generate clean code? In our study, we have found positive and negative answers to the question, varying with respect to different aspects of source code quality and the tools. Based on this experience we reported four anti-patterns that describe the encountered issues, along with possible solutions towards the development of better GUI builders. The reported anti-patterns further provide some basic directions to GUI developers/designers for the selection of appropriate GUI builders.

How can you (tool-smiths) develop better GUI Builders?

- *Generate meaningful names.* To this end, *avoid naming patterns that combine type information with number series* because the resulting names will not reveal the purpose of the named elements. *Employ naming patterns that exploit information like labels or textual descriptions*, provided by the GUI developer for the characterization of the named elements. *Prompt the GUI developer* to provide *meaningful content* for the GUI elements, *reward him* for that and possibly *check the quality of the provided content* on the fly with respect to given linguistic anti-patterns [3].
- *Do not bloat the generated code with comments.* If you feel that the generated code should include a comment, *spend time to improve the code generation process*, to produce *cleaner code* that *does not need explanations*. Generate comments that facilitate integration, extension and maintenance.
- *Do not generate god classes* that *mix widget configuration and event handling responsibilities*. To this end, you can follow a model-view-controller approach. *Considering the event handling controllers*, a possible challenge is to develop a tool that allows the GUI developer to *customize their granularity* according to the specificities of the GUI that he/she develops. Another interesting feature would be to enable the *reuse of views or view parts*, possibly via some kind of parameterization.
- *Avoid the generation of widget configuration clones. To achieve this,* organize the generated code in terms of small interpreter methods *that initialize different types of widgets, based on respective* widget property specifications. Benefit from features of the underlying technology, *like GUI model specification facilities, paired with technology-specific GUI initialization interpreters.*

How can you (GUI designers/developers) select appropriate GUI Builders?

- Select GUI builders based on both the *provided GUI design features* and the *quality of the generated GUI implementation.*
- To assess the quality of the GUI implementation consider *the generated widget names.* Look for GUI builders that generate widget names which reflect the purpose of the widgets in the GUI, make meaningful distinctions between widgets, are search friendly, pronounceable, and so on.
- Take into account *the generated GUI documentation.* Select GUI builders that generate minimal documentation which facilitates the GUI integration, extension and maintenance.
- Pay attention to the *size* and the *complexity* of *the generated GUI implementation.* Look for GUI builders that separate widget configuration from event handling. Select GUI builders that generate clone free widget configuration code.

Future perspectives: In our study, we focused on the quality of the code that is generated by GUI builders. Assessing the provided GUI design features is also a primary concern that can be considered as future work. Several interesting issues can be investigated like the variety of widgets that are offered, support for multiple widget APIs, GUI internalization, integration with available testing frameworks, and so on. For instance, the tools that we investigated in our study provide means for GUI internalization. Although the details on how this is done vary between the tools, the main idea is common, i.e., to specify widget names in a language-specific external properties file that can be easily changed. An issue in this approach is that the GUI developer has to perform most of the work that concerns the names translation in multiple languages. Hence, automating the *identification of equivalent terms* in different languages could be a useful feature. Finally, it would be interesting to study further combinations of tools and technologies, like Web development frameworks/IDEs, Mobile App development frameworks/IDEs, etc.

ACKNOWLEDGMENTS

We want to thank our shepherd, Ralf Laue, for his valuable comments, suggestions, and overall guidance in the preparation of this paper. We would also like to thank his students attending the course "Patterns and Pattern Languages" at the University of Applied Sciences Zwickau for their remarks and the suggested future work issues.

REFERENCES

[1] Surafel Lemma Abebe, Sonia Haiduc, Paolo Tonella, and Andrian Marcus. 2009. Lexicon Bad Smells in Software. In *Proceedings of the 16th IEEE Working Conference on Reverse Engineering (WCRE)*. 95–99.
[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the Joint 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering and 15th European Software Engineering Conference (FSE/ESEC)*. 38–49.
[3] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic Antipatterns: What They Are and How Developers Perceive Them. *Empirical Software Engineering* 21, 1 (2016).
[4] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.
[5] William Brown, Raphael Malveau, Hays McCormick, and Thomas Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley.
[6] Raymond P. L. Buse and Westley Weimer. 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering* 36, 4 (2010), 546–558.
[7] Simon Butler, Michel Wermelinger, and Yijun Yu. 2015. Investigating Naming Convention Adherence in Java References. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 41–50.
[8] CheckStyle Team. 2017. Checkstyle. checkstyle.sourceforge.net/index.html.
[9] Jehad Al Dallal. 2015. Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review. *Information and Software Technology* 58, 0 (2015), 231 – 249.
[10] Martin Fowler. 2009. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley.
[11] Neil B. Harrison. 2004. Advanced Pattern Writing Patterns for Experienced Pattern Authors. Avaya, inc..
[12] Yuki Kashiwabara, Yuya Onizuka, Takashi Ishio, Yasuhiro Hayase, Tetsuo Yamamoto, and Katsuro Inoue. 2014. Recommending Verbs for Rename Method

Using Association Rule Mining. In *Proceedings of the 21st IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 323–327.

[13] Ralf Laue. 2017. Anti-Patterns in End-User Documentation. In *Proceedings of the 22nd ACM European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 20:1–20:11.

[14] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *Proceeedings of the 14th IEEE International Conference on Program Comprehension (ICPC)*. 3–12.

[15] Robert C. Martin. 2009. *Clean Code - A Handbook of Agile Software Craftsmanship*. Prentice Hall.

[16] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.

[17] Gerard Meszaros and Jim Doble. 1997. Pattern Languages of Program Design 3. Chapter A Pattern Language for Pattern Writing, 529–574.

[18] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study. In *Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST)*. 5–14.

[19] Trygve Reenskaug. 1979. A Note on DynaBook Requirements. Xerox PARC.

[20] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[21] Michael Smit, Barry Gergel, H. James Hoover, and Eleni Stroulia. 2011. Code Convention Adherence in Evolving Software. In *Proceedings of the 27th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 504–507.

[22] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 43–52.

[23] Markus Völter. 2003. A Catalog of Patterns for Program Generation. In *Proceedings of the 8th ACM European Conference on Pattern Languages of Programms (EuroPLoP)*. 285–320.

[24] Tim Wellhausen and Andreas Fiesser. 2012. How to Write a Pattern?: A Rough Guide for First-time Pattern Authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 5:1–5:9.