Multi-Level Slicing of Automatically Generated GUIs

Christos Petridis

Diploma Thesis

Supervisor: Apostolos Zarras

Ioannina, May, 2018

ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA

Prologue

The present diploma thesis entitled “Multi-Level Slicing of Automatically Generated GUIs” was drafted in the Department of Computer Science and Engineering, University of Ioannina.

I want to thank Apostolos Zarras, Associate Professor at the department, for his guidance and valuable help, as well as my parents for supporting me throughout my studies.

May 2018

Christos Petridis

Abstract

Program slicing is a decomposition technique that elides program components not relevant to a chosen computation, referred to as slicing criterion. We introduce a new slicing technique especially for code generated automatically by GUI-Builders, named Widget Slicing. We analyze the basic structure, purpose and potential of Widget Slices, as well as provide an independent and elegantly designed tool for finding these kinds of slices. Finally, we put our tool to test, using cases from some of the most popular GUI-Builders at present.

Keywords: program slicing, widget, independent tool

Contents Table

1. Introduction

Clean code is a key issue in the development and the overall life-cycle of a project. Robert Martin [Martin 2009] performed a detailed analysis of several aspects considering code quality and showed how important it is for testing, maintaining and comprehending code.

In this Thesis we focus on GUIs, as their implementation is inherently long and complex. This makes it hard to easily identify and link a GUI element to the code affecting it. So, tasks like testing or maintenance become unpleasant and difficult. Mamalis [Mamalis 2017] showcased this problem in the particular case of automatically generated GUIs using some of the most popular GUI-Builders at present. Though we focus more on automatically generated code for testing, our ideas can also be applied to hand written GUIs.

A solution regarding this problem would be to refactor the GUI code. Kollias [Kollias 2018] presented a set of different steps one can follow in order to improve code quality, such as making structured widget specifications by creating new methods for constructing widgets or extracting widget specific classes for a more fine grained structure. However, implementing those steps manually would require someone to spend far more time than he should (or has) for this kind of job.

This diploma Thesis focuses on the first step needed for automating the processes mentioned, which is gathering all the necessary information concerning widgets, in a structured way. We base our idea on the technique of program slicing introduced by Mark Weiser [Weiser 1984] and propose a new kind of slicing, named *Widget Slicing*. Furthermore, we provide "JWiSH" [Petridis 2018], a highly extendable tool for implementing this technique by creating what we call *Widget Dependence Graph* (WDG) from now on and using it for finding widget slices.

The remainder of the paper is as follows. In chapter 2, we present the background and the related work that led to our idea. In chapter 3, we focus on the theoretical aspects of

Widget Slicing and explain in detail its structure and meaning. In chapter 4, we take a closer look at the architecture, design and potential of JWiSH. In chapter 5, we test our tool and present some statistics about its performance. Finally in chapter 6, we suggest possible future work and review our results.

2. Background

As mentioned above, Mamalis [Mamalis 2017] evaluated the quality of code generated by GUI-Builders. He focused on important issues like meaningful names, comments and the design/implementation of classes and methods. He used four popular GUI-Builders (SWT, NetBeansEditor, JFormDesinger and WindowBuilder) to replicate the basic structure of the Eclipse IDE as showed in Figure 2.1 beneath.

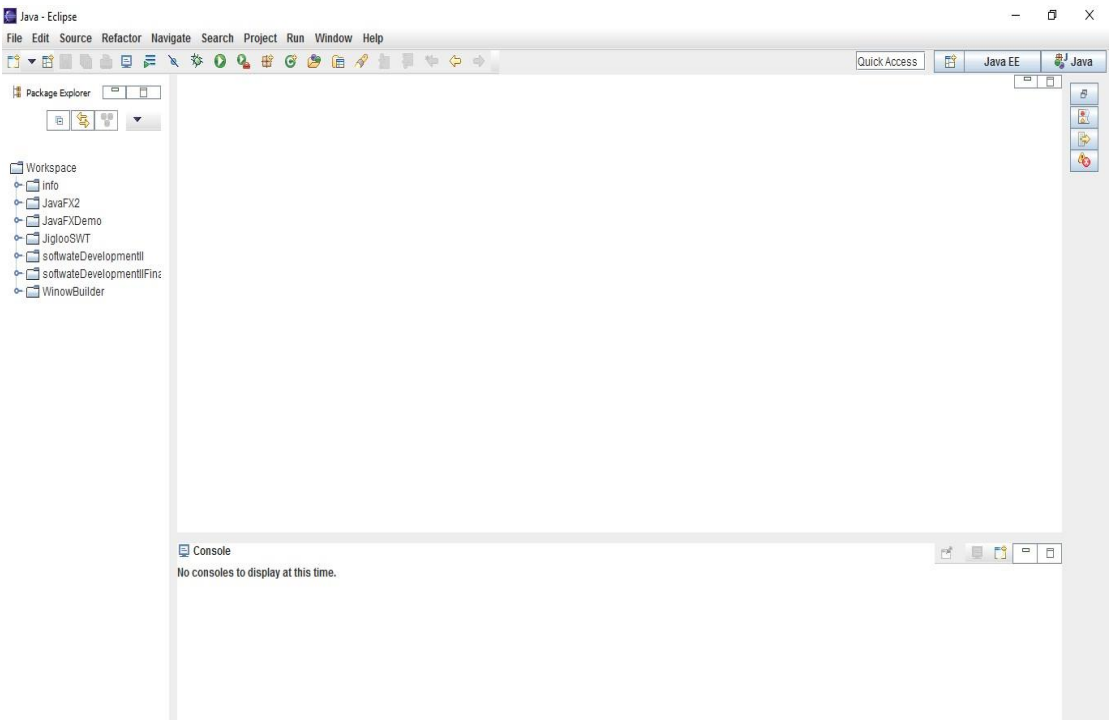

Figure 2.1-Replica of Eclipse IDE generated using WindowBuilder.

Then, Mamalis analyzed the source code produced and found that in all of the cases basic principles defined by Martin [Martin 2009] were violated. Such principles include small methods and single method responsibilities (Table 2.1), small classes with single responsibilities (Table 2.2) and code duplication (Table 2.3).

Method name	Case study	Size
EclipseGUI( )	WindowBuilder	4235
initComponents( )	NetBeansEditor	5550
initComponents( )	JFormDesinger	4172
open( )	SWT	5323

Table 2.1 - Methods that violate both the size and single responsibility principle. [Mamalis 2017]

Case Study	Size	Methods
WindowBuilder	4324	3
NetBeansEditor	8123	6
JFormDesinger	6572	3
SWT	5338	4

Table 2.2 - Classes that violate both the size and single responsibility principle. [Mamalis 2017]

Case study	% of Duplicated Code
WindowBuilder	81%
NetBeansEditor	40,5%
JFormDesinger	30,6%
SWT	81%

Table 2.3 – Amount of duplicated code produced. [Mamalis 2017]

By observing those results it becomes clear that automatically generated code from GUI-Builders can be of poor quality and creates anti-patterns. Kollias [Kollias 2018] tried to resolve that by applying a set of different refactoring techniques using the cases above. Such techniques include separating event handling from widget initialization, creating structured widget specifications by following a set of given steps and extracting classes for specific widget construction.

The example detailed in the sequel is based on code generated by the WindowBuilder tool.

In figures 2.1, 2.2 we display a code sample of widget configuration clones to get a glimpse of how complex and unorganized the code is. In figures 2.3, 2.4 and 2.5 after

having performed refactoring using the steps suggested for the structured specifications of widgets [Kollias 2018], we can observe how smaller, easier to read and understand and in general, cleaner the code is.

In figure 2.1 we see code generated for menu items and in figure 2.2 for buttons. In figure 2.3 we can see a class defined for the encapsulation of widget properties. Widget creation methods can use objects of the class to get all the widget's properties. In figure 2.4, factory class is displayed. This class hosts the widget creation methods, so that they do not mess with the GUI code. In figure 2.5 we can observe a case of creating 16 menu items using the new defined structures mentioned.

The size of the code is significantly reduced and now we can distinguish all the different parts that make up the source code and their responsibilities.

```
JMenuItem mntmNext = new JMenuItem("Next Annotation");
mntmNext.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
mntmNext.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/next_annotation.gif")));
mntmNext.setEnabled(false);
mnNavigate.add(mntmNext);
```

```
JMenuItem mntmPrevious = new JMenuItem("Previous Annotation");
mntmPrevious.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
mntmPrevious.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/prev_annotation.gif")));
mntmPrevious.setEnabled(false);
mnNavigate.add(mntmPrevious);
```

```
JMenuItem mntmLastEditLocation = new JMenuItem("Last Edit Location");
mntmLastEditLocation.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
mntmLastEditLocation.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/last_edit_pos.gif")));
mntmLastEditLocation.setEnabled(false);
mnNavigate.add(mntmLastEditLocation);
```

Figure 2.1 – An example of menu items configuration clones. [Kollias 2018]

```
JButton btnNewButton_2 = new JButton("");
btnNewButton_2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
btnNewButton_2.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/problems_view.gif")));
btnNewButton_2.setBounds(0, 24, 35, 23);
panel_3.add(btnNewButton_2);
```

```
JButton button_3 = new JButton("");
button_3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
button_3.setBackground(Color.WHITE);                                                    ;
button_3.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/minimize.gif")));
button_3.setBounds(1030, 0, 31, 21);
panel_2.add(button_3);
```

```
JButton button_4 = new JButton("");
button_4.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});                                                                                     ));
button_4.setBackground(Color.WHITE);
button_4.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/thin_max_view.gif")));
button_4.setBounds(1060, 0, 31, 21);
panel_2.add(button_4);
```

Figure 2.2 – An example of buttons configuration clones. [Kollias 2018]

```
class ComponentProperties {

    String name, iconName;
    boolean enabled;
    ActionListener actionListener;
    JComponent container;
    List<Integer> bounds;
    Color backgroundColor;


    ComponentProperties(String name, boolean enabled, List<Integer> bounds, String iconName, Color backgroundColor,
            JComponent container, ActionListener actionListener) {
        this(name, enabled, bounds, iconName, container, actionListener);
        this.backgroundColor = backgroundColor;
    }

    ComponentProperties(String name, boolean enabled, List<Integer> bounds, String iconName, JComponent container,
            ActionListener actionListener) {
        this(name, enabled, bounds, container, actionListener);
        this.iconName = iconName;
    }

    ComponentProperties(String name, boolean enabled, List<Integer> bounds, JComponent container, ActionListener actionListener) {
        this(name, enabled, container, actionListener);
        this.bounds = bounds;
    }

    ComponentProperties(String name, boolean enabled, String iconName, JComponent container, ActionListener actionListener) {
        this(name, enabled, container, actionListener);
        this.iconName = iconName;
    }

    ComponentProperties(String name, boolean enabled, JComponent container, ActionListener actionListener) {
        this.name = name;
        this.enabled = enabled;
        this.actionListener = actionListener;
        this.container = container;
    }
}
```

Figure 2.3 – Parameter object class.

```
class ComponentFactory {

    void createMenuItem(ComponentProperties menuItemProperties) {
        JMenuItem menuItem = new JMenuItem(menuItemProperties.name);
        menuItem.setEnabled(menuItemProperties.enabled);
        if (menuItemProperties.iconName != null)
            menuItem.setIcon(new ImageIcon(EclipseGUI.class.getResource(menuItemProperties.iconName)));
        menuItem.addActionListener(menuItemProperties.actionListener);
        menuItemProperties.container.add(menuItem);
    }

    void createButton(ComponentProperties buttonProperties) {
        JButton button = new JButton(buttonProperties.name);
        button.setEnabled(buttonProperties.enabled);
        List<Integer> bounds = buttonProperties.bounds;
        button.setBounds(bounds.get(0), bounds.get(1), bounds.get(2), bounds.get(3));
        if (buttonProperties.iconName != null)
            button.setIcon(new ImageIcon(EclipseGUI.class.getResource(buttonProperties.iconName)));
        if (buttonProperties.backgroundColor != null)
            button.setBackground(buttonProperties.backgroundColor);
        button.addActionListener(buttonProperties.actionListener);
        buttonProperties.container.add(button);
    }

}
```

Figure 2.4 – Factory class for menu items and buttons.

```
ComponentProperties[] mnNewMenuItemsProperties = {
        new ComponentProperties("Java Project", true, "/images/newjprj_wiz.gif", mnNew, null),
        new ComponentProperties("Project...", true, "/images/new.gif", mnNew, null),
        new ComponentProperties("Package", true, "/images/new_package.gif", mnNew, null),
        new ComponentProperties("Class", true, "/images/new_class.gif", mnNew, null),
        new ComponentProperties("Interface", true, "/images/newint_wiz.gif", mnNew, null),
        new ComponentProperties("Enum", true, "/images/newenum_wiz.gif", mnNew, null),
        new ComponentProperties("Annotation", true, "/images/newannotation_wiz.gif", mnNew, null),
        new ComponentProperties("Source Folder", true, "/images/newpackfolder_wiz.gif", mnNew, null),
        new ComponentProperties("Java Working Set", true, "/images/java_working_set.png", mnNew, null),
        new ComponentProperties("Folder", true, "/images/newfolder_wiz.gif", mnNew, null),
        new ComponentProperties("File", true, "/images/new_con.gif", mnNew, null),
        new ComponentProperties("Untitled Text File", true, "/images/new_untitled_text_file.gif", mnNew, null),
        new ComponentProperties("JUnit Test Case", true, "/images/test.gif", mnNew, null),
        new ComponentProperties("Task", true, "/images/new_con.gif", mnNew, null),
        new ComponentProperties("Example...", true, "/images/new.gif", mnNew, null),
        new ComponentProperties("Other...", true, "/images/new.gif", mnNew, null) };

for (ComponentProperties menuItemProperties : mnNewMenuItemsProperties)
    componentFactory.createMenuItem(menuItemProperties);
```

Figure 2.5 – Menu with 16 menu items that are created and configured in a 'for' loop.

Applying such refactoring techniques can improve code quality to a great extent. However, the amount of time spent and difficulty of executing those tasks manually often counteracts the benefits received. For managing these tasks automatically we concluded that two steps are needed:

- Gathering all the information regarding the widgets that make up a GUI, their properties and the relation between them, in a structured meticulous way.
- Analyzing that information so that we can determine how to organize it and extract the properties we need for performing various refactoring techniques.

In this Thesis, we will focus on the first step.

3. Widget Slicing

3.1 Basic Concepts

The original idea of program slicing comes from Mark Weiser [Weiser 1984], it is a technique to decompose programs by analyzing their data and control flow. It strips away all the program components that are irrelevant to a given computation, referred to as slicing criterion, and computes the program slice. Since then, many slicing-based techniques have appeared, each serving a different purpose but mainly refactoring, debugging and testing. A fairly comprehensive survey of methods is presented in [Josep Silva 2012]. A simple example of the original static slicing technique is presented in Table 3.1 below.

Original Program	Slicing criterion	Slice
a = 23 b = 42 c = b + 2	c	b = 42 c = b + 2

Table 3.1 – Static slicing using name variable 'c' as slicing criterion.

So, to enable the refactoring of GUI implementations, we used the idea of program slicing as our guide and proposed a new slicing technique, naming it *Widget Slicing*. Roughly speaking, a widget slice consists of those program statements that affect the state of a widget. For example, in Figure 3.1 we can see what the widget slice would be using slicing criterion *<mntUndoTyping>.* We distinguish two kinds of slices, **Primitive Widget Slices** (PWS) and **Composite Widget Slices** (CWS). The extraction of widget slices is based on the *Widget Dependence Graph* which we will analyze in detail in

Section 3.2. Following, in Section 3.3 we discuss each kind in further detail and describe how to utilize the WDG.

```
Menu menu_2 = new Menu(mntmEdit);
mntmEdit.setMenu(menu_2);

MenuItem mntmUndoTyping = new MenuItem(menu_2, SWT.NONE);
mntmUndoTyping.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
    }
});
mntmUndoTyping.setImage(SWTResourceManager.getImage(EclipseGUI.class, "/images/undo_edit.gif"));
mntmUndoTyping.setEnabled(false);
mntmUndoTyping.setText("Undo Typing");

MenuItem mntmRedo = new MenuItem(menu_2, SWT.NONE);
mntmRedo.addSelectionListener(new SelectionAdapter() {
    @Override
    public void widgetSelected(SelectionEvent e) {
    }
});
mntmRedo.setImage(SWTResourceManager.getImage(EclipseGUI.class, "/images/redo_edit.gif"));
mntmRedo.setEnabled(false);
mntmRedo.setText("Redo");

new MenuItem(menu_2, SWT.SEPARATOR);
```

Figure 3.1 – Widget Slice for slicing criterion <mntUndoTyping>.

3.2 WDG

In general, most slicing techniques rely on a variety of different concepts for constructing slices. Such concepts are a program's Control Flow Graph, Dependence Graph, System Dependence Graph, Abstract Syntax Tree and so on. To facilitate widgets and slices, we propose a new dependence graph especially for GUI elements called Widget Dependence Graph (WDG).

A WDG relates widgets with other widgets and allows us to group them together because it has information about the connection between them. Each node in the WDG represents a widget used in GUI. There are two kinds of nodes, Primitive and Composite. Composite nodes are used to represent composite widgets, namely widgets that can contain other widgets like menus or panels and Primitive represent the rest. An edge from a source node to a destination node specifies that the widget represented by the source node contains the widget represented by the destination node. Therefore, the source node is a Composite node and the destination node is a Primitive or another Composite node. In Figure 3.2 we can see how the WDG looks for the simple example of a menu adding a menu item to itself.

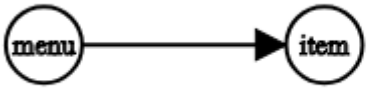

Figure 3.2 – A simple example of dependence.

Since Composite nodes can contain other composites, a multi-level graph can be established. Primitive nodes start at level 0 and the Composite node's level is determined by the highest level node that it contains. Notice that a Composite node can contain nodes of different levels. Figure 3.3, contains part of a code generated using Swing and figure 3.4, shows the WDG that corresponds to that code part.

```
JMenu mnNavigate = new JMenu("Navigate");
menuBar.add(mnNavigate);

JMenuItem mntmGoInto = new JMenuItem("Go Into");
mntmGoInto.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
mntmGoInto.setEnabled(false);
mnNavigate.add(mntmGoInto);

JMenu mnGoTo = new JMenu("Go To");
mnNavigate.add(mnGoTo);

JMenuItem mntmBack_1 = new JMenuItem("Back");
mntmBack_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
mntmBack_1.setEnabled(false);
mnGoTo.add(mntmBack_1);

JMenuItem mntmForward_1 = new JMenuItem("Forward");
mntmForward_1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
mntmForward_1.setEnabled(false);
mnGoTo.add(mntmForward_1);

JMenuItem mntmUpOneLevel = new JMenuItem("Up One Level");
mntmUpOneLevel.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
```

Figure 3.3 – Part of code generated with Swing.

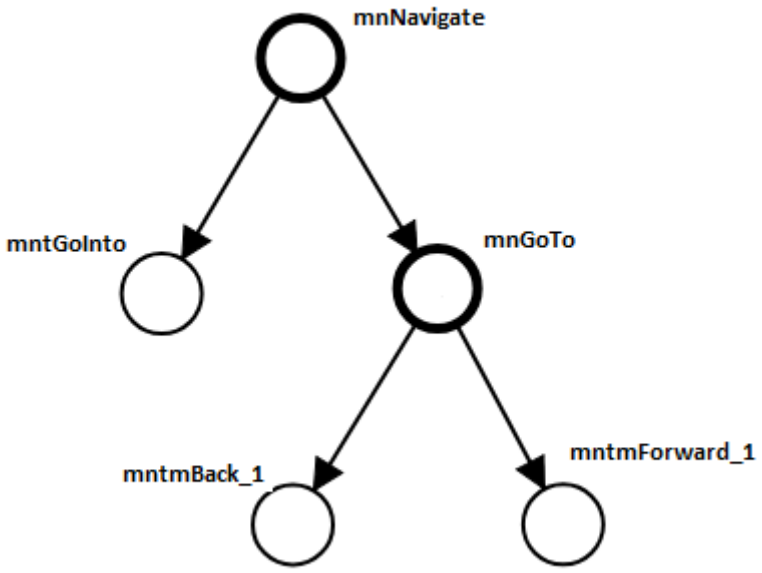

Figure 3.4 – Partial WDG. Bold border nodes are Composite.

In figure 3.3, the widgets that participate in the example are marked with red (solid lines) and the statements that add a widget to another are marked with green (dashed lines). Observe in figure 3.4, *mnGoTo* is level 1 and *mnNavigate* level 2. Also, *mnNavigate* contains nodes of different levels.

It is important to mention that the WDG can in some cases be a disconnected graph with a small amount of sub-graphs and there can also be isolated nodes. Furthermore, leaf nodes always have a level of 0 and each sub-graph has potentially a different max level.

Node Information: Nodes in the WDG, no matter their kind, hold specific information that we deemed would be useful when trying to refactor GUI source code. This information includes the statements that affect the state of its widget (as in figure 3.1), the name of it, its level and the type of the widget. Each piece of information allows you to find slices using different criteria, depending on what refactoring technique you want to use.

Now, using the information available in the WDG about the relations between widgets and the information stored inside its nodes we can determine widget slices and complete the first step.

3.3 Widget Slices

As mentioned in section 3.1, widget slices can be separated into two kinds, primitive and composite. We will analyze each type and the role it plays separately.

Primitive Widget Slices are single nodes in the WDG. They are constructed in relation to the Primitive nodes of the graph and the information they hold about the statements affecting the widgets. These slices are computed by using the name of a widget as the slicing criterion. For example, if we use slicing criterion <widget-name> then the node representing that widget would return us the slice that corresponds to it, like the example in figure 3.1. In figure 3.5 we can see a schematic representation of the slice within the WDG of figure 3.4, using criterion <mntmForward_1>.

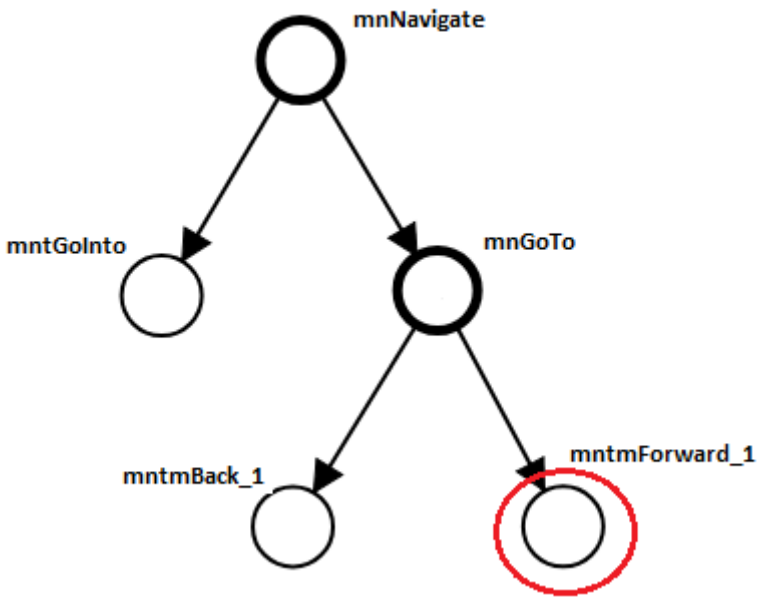

Figure 3.5 – Visualized Widget Slice of mntmForward_1.

In the case of **Composite Widget Slices** things are more complicated. Composite slices are essentially sub graphs of the WDG. They consist of unions of widgets slices, whether they are PWS or other CWS. Though De Lucia [De Lucia et al. 2003] showed that in theory -if we rigidly stick to the definition of slice- unions of slices are not slices, there are conditions like in the case of *Union Slicing* or *Simultaneous Slicing* [Josep Silva 2012] that allow the formation of such slices. Widget Slices are part of those exceptions.

For example, if we use criterion <mnNavigate> from figure 3.4, the CWS returned would consist not only by the statements that affect the state of *mnNavigate* (circle number 1, green), but also the PWS of *mntmGoInto* (circle number 2, purple) and CWS of *mnGoTo* (circle number 3, blue). It would look something like figure 3.6 below. Notice that the second line is not part of the slice.

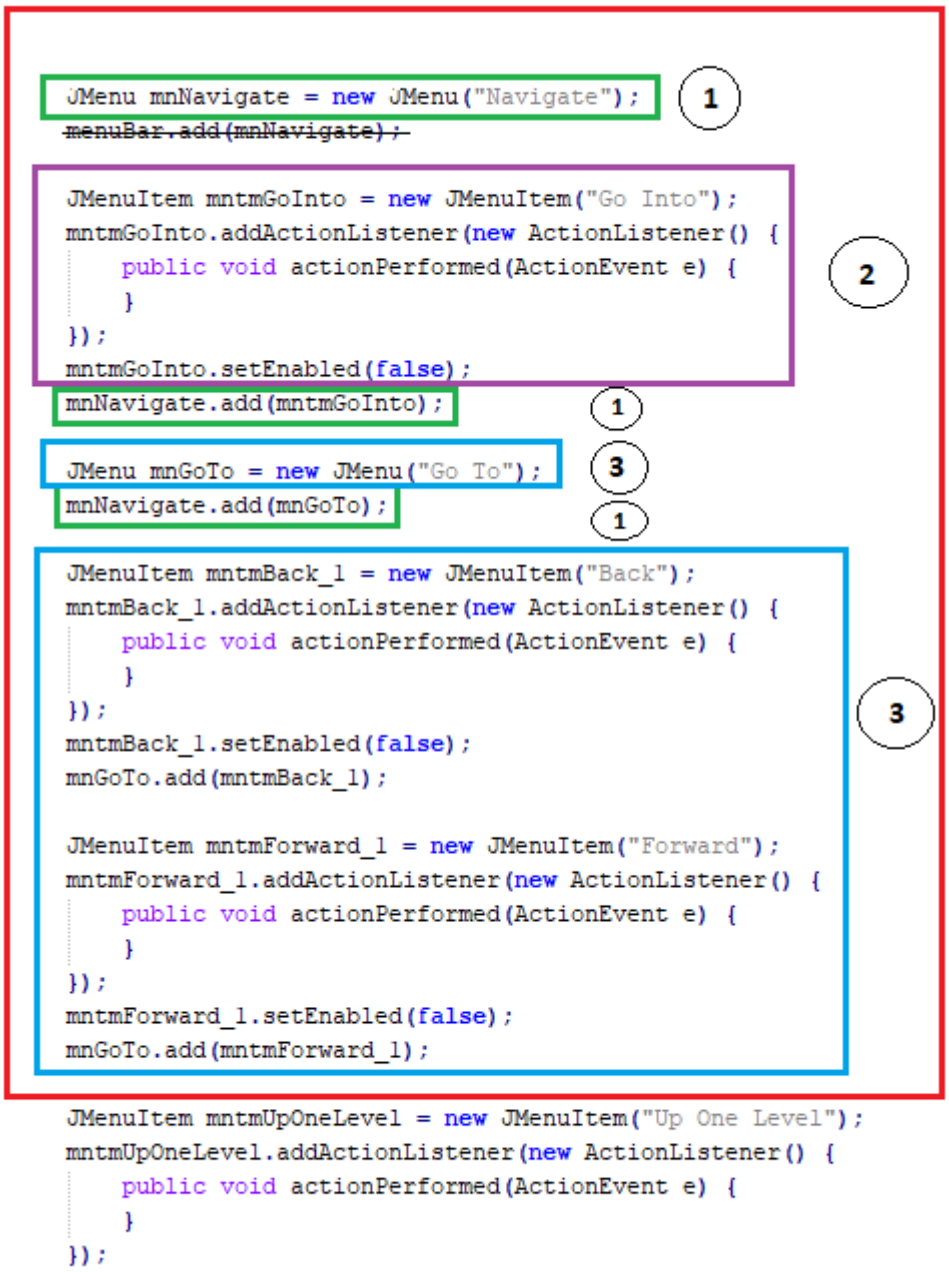

Figure 3.6 – CWS of mnNavigate.

It's important to mention here that *mntmGoInto* and *mnGoTo* are part of the widget slice of *mnNavigate*, because *mnNavigate* contains those two widgets. In general, if widget **P** has a widget slice **WSP** and widget **C** has a widget slice **WSC**, then if P contains C, WSP also contains WSC.

Other criteria except for a widget's name include the level of a node and the type of the widget a node contains. Slicing with criterion <level> for example, would make a slice consisting of all the widgets that match exactly that level. This can be useful to isolate all the Primitive widgets or when combined with other criteria like <type>, to create, for example, structured specifications of widgets as mentioned in [Kollias 2018]. We can see in figure 3.7, part of how the CWS would look if level was set to 1 and in figure 3.8, how the CWS would look using criterion <type = menuItem>. Both examples are taken from the JFormDesinger case.

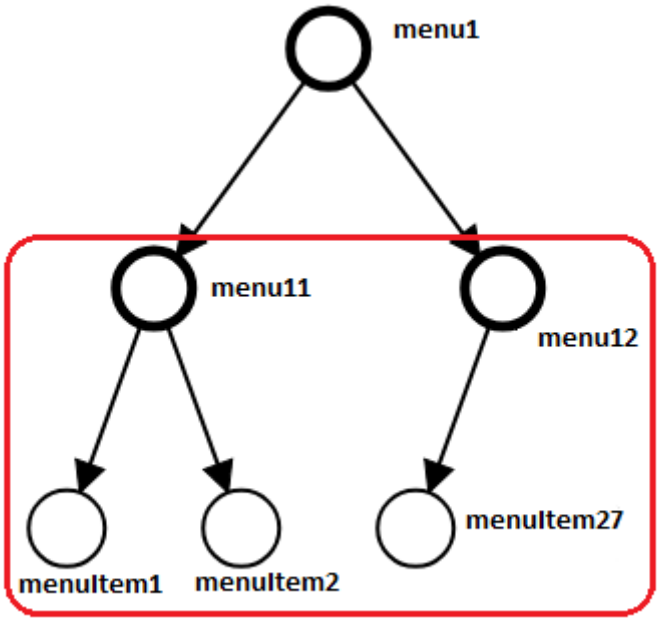

Figure 3.7 – Widget Slice for criterion <1>.

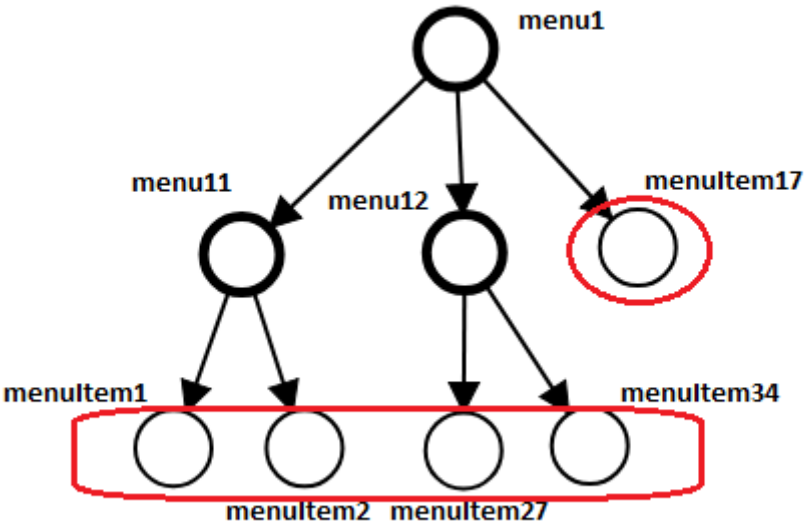

Figure 3.8 – Widget Slice for criterion <menuItem>.

Grouping together widgets like this allows us to perform all the different refactoring techniques mentioned in [Kollias 2018]. For example, we can take the slice of a high level widget and extract it in a class along with its contents or we can make a slice consisting of all the widgets of a specific type and analyze it to construct methods for their initialization (e.g. createButton() in figure 2.4). There are as many possibilities as there are ways of slicing.

4. JWiSH

Now that we have set up the theoretical background of widget slicing, we are going to analyze the tool we developed for finding widget slices. We named the tool JWiSH, from *Widget Slice Highlighting*, as in highlighting the part of the WDG that satisfices the slicing criterion. JWiSH is elegant and highly extendable by design. It can potentially analyze the GUI source code of any project regardless of the programming language or the GUI-Builder used. To do that, it was designed in a way that allows third party developers to add new features and increase its capabilities. Those features include the addition of new parsers, either already existing or completely new and the addition of new widget APIs. We are going to present in a more descriptive way the overall architecture and design of the tool, as well as provide the necessary steps for adding these features and the parts that require caution when doing so.

4.1 Architecture

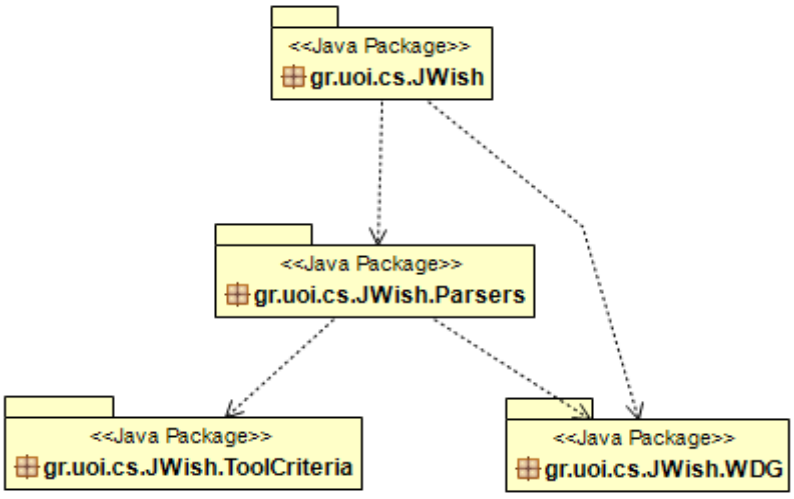

Figure 4.1 – Architecture of JWiSH.

In figure 4.1 we can see the packages that comprise JWiSH. Package "Parsers" holds the functions of the parsers, package "ToolCriteria" holds the widget APIs and package "WDG" holds the implementation of the Widget Dependence Graph.

4.2 Detailed Design

Note: Because of the complexity of the UML diagrams, we are going to show each package separately and focus only on the important aspects of the tool (meaning trivial dependencies may not be included).

Parsers: In figure 4.2 we can see the UML diagram of the package containing the parser's utilities. The abstract class *Parser* is responsible for finding the widget slices by parsing and analyzing the GUI files. We can add a concrete parser and increase the capabilities of the tool by implementing a single method called "findSlices()". As we can see, we have already added ASTParser to JWiSH. ASTParser is a Java language parser for creating abstract syntax trees (AST), meaning trees that represent the syntactic structure of source code. Each node in the AST denotes an entity of the code, like if-statements or variable declarations.

When adding a new parser to JWiSH there is a critical point we must take under consideration. That is the entities of the code that we should examine, because the information we need is only found in specific parts of the code. For example, a method declaration is of no use to us in the process of making widget slices. The entities of a program to be considered are variable declarations, assignments, method invocations and possibly qualified names (depending on the GUI-Builder used). Each one should be examined separately for extracting the necessary information from the source code.

In cases where the parser, lacks or doesn't offer an easy way of extracting important information about the widgets, like variable types, we can use a supporting class to help us acquire that information. In the case of ASTParser we couldn't access the type of member variables or Constructor names, so we created and used the class "StringAdministrator" to acquire that information by managing Strings.

Another issue to consider is the way each widget API employs for adding one widget to another. We will discuss this issue in further detail when we'll talk about the "ToolCriteria" package, but it is important to mention here that method invocations and

Constructor parameters should be checked for identifying when an addition event like that occurs and if so, to determine which widget depends on another. This check should be done for each of the variable declaration, assignment, method invocation and qualified name entities.

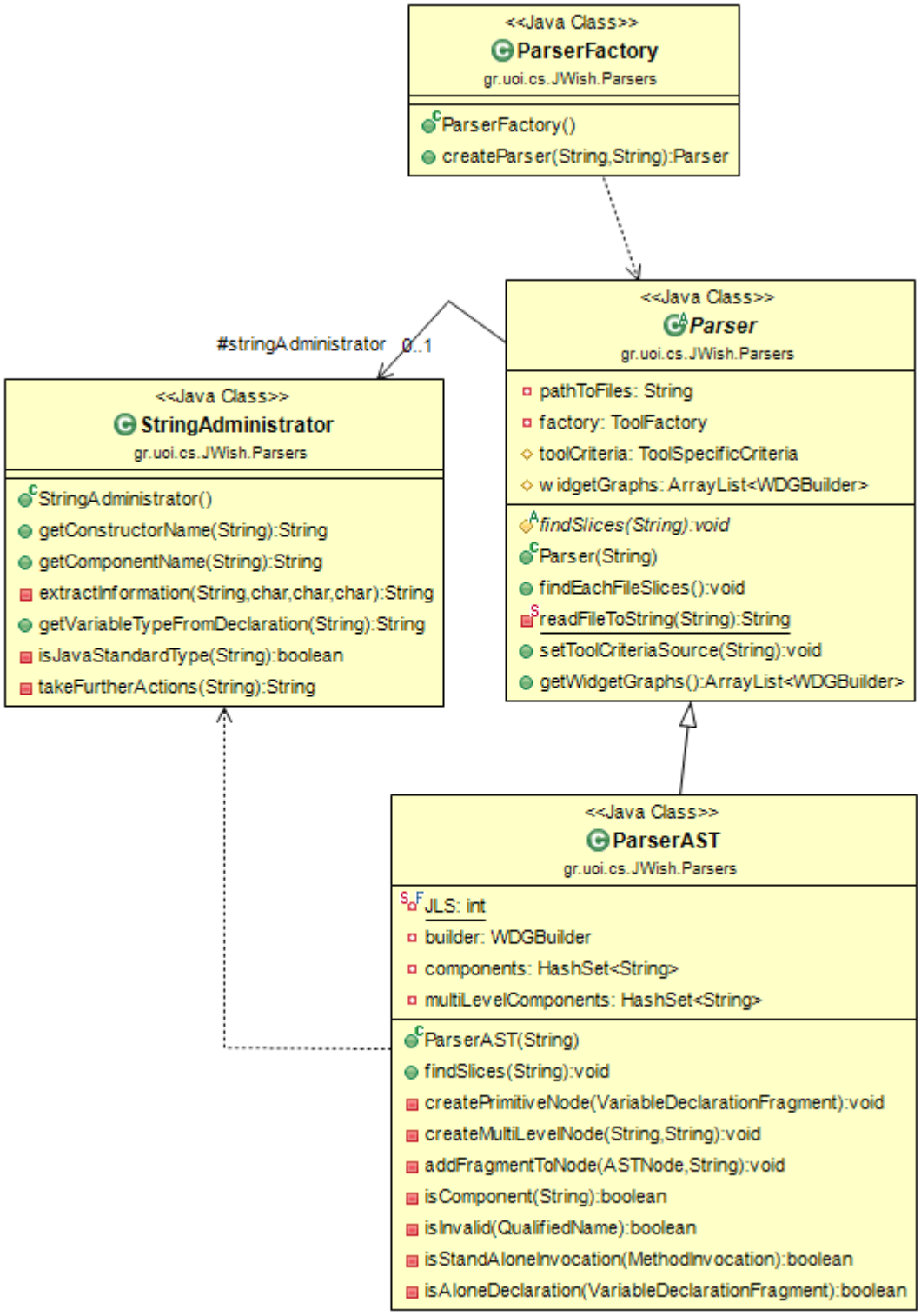

Figure 4.2 – UML Diagram of the parsers package.

GUI-Builders: In figure 4.3 we can see the UML diagram of the "ToolCriteria" package. To add a new widget API to JWiSH there are two things that we should determine. We need to search the API and identify the following:

- Class names of all the Widget types supported by the API (e.g. Menu, Button), so we don't concern ourselves with valueless slices from non-widget variables that may exist in our code.
- All the possible ways the widget API provides for adding one widget to another. In Swing for instance, the invocation of the method "add" plays that role (e.g. menu.add(menuItem)).

So by criteria, we mean the widget class names and the different ways of relating widgets. "ToolSpecificCriteria" holds this information that is later used by the parser to help construct the slices. Because of that, the two packages, "ToolSpecificCriteria" and "Parsers", are strongly connected.

We use the method "initializeWidgetNames()" to specify the class names and the method "initializeComponentAdditionEvent()" to specify the different ways of relating widgets. We have already added Swing and SWT to JWiSH. Notice that the first uses method invocations for widget addition, whereas the other does it through Constructors. So, when we add a new parser we should determine where (method invocations, Constructors etc.) and how (Support class/Parser) to get the right information to compare with the tool criteria. Also, the parser should be independent of the way the API provides for widget addition.

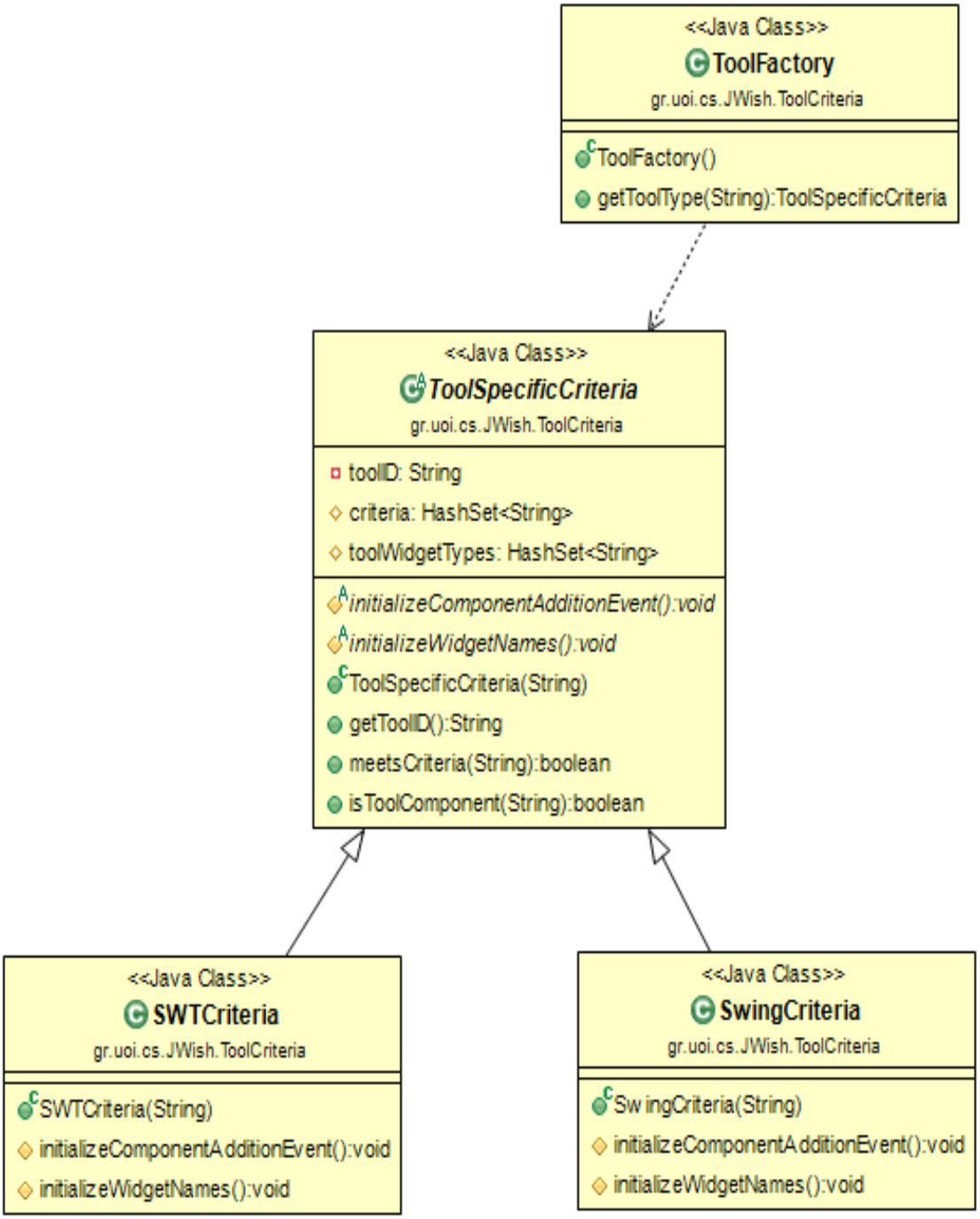

Figure 4.3 – UML Diagram of ToolCriteria package.

WDG: In figure 4.4 we can see the UML diagram of the package holding the basic structure of the WDG. The class "WDGBuilder" is used by the parser to construct the graph. From the "WDG" package, only "WDGBuilder" is able to interact with the parser. This makes the construction of the WDG independent from the programming language used, because the WDG builder doesn't need to know where the node information we provide him, came from. In fact, here we applied the concept of the Builder Design Pattern [GoF 1994].

Observe the Composite Design ([GoF 1994]) between the classes "WDGNode", "CompositeNode" and "PrimitiveNode", as "CompositeNode" has a list with objects from the abstract class "WDGNode". This composite pattern helps us establish multilevel widgets.

Because of the fact that statements that belong to a single widget slice can be found scattered across the file, like the statements of mnNavigate in figure 3.6 (circle number 1 statements), every node has a list of code fragments representing each statement. We use the class "SliceFragment" for that purpose.

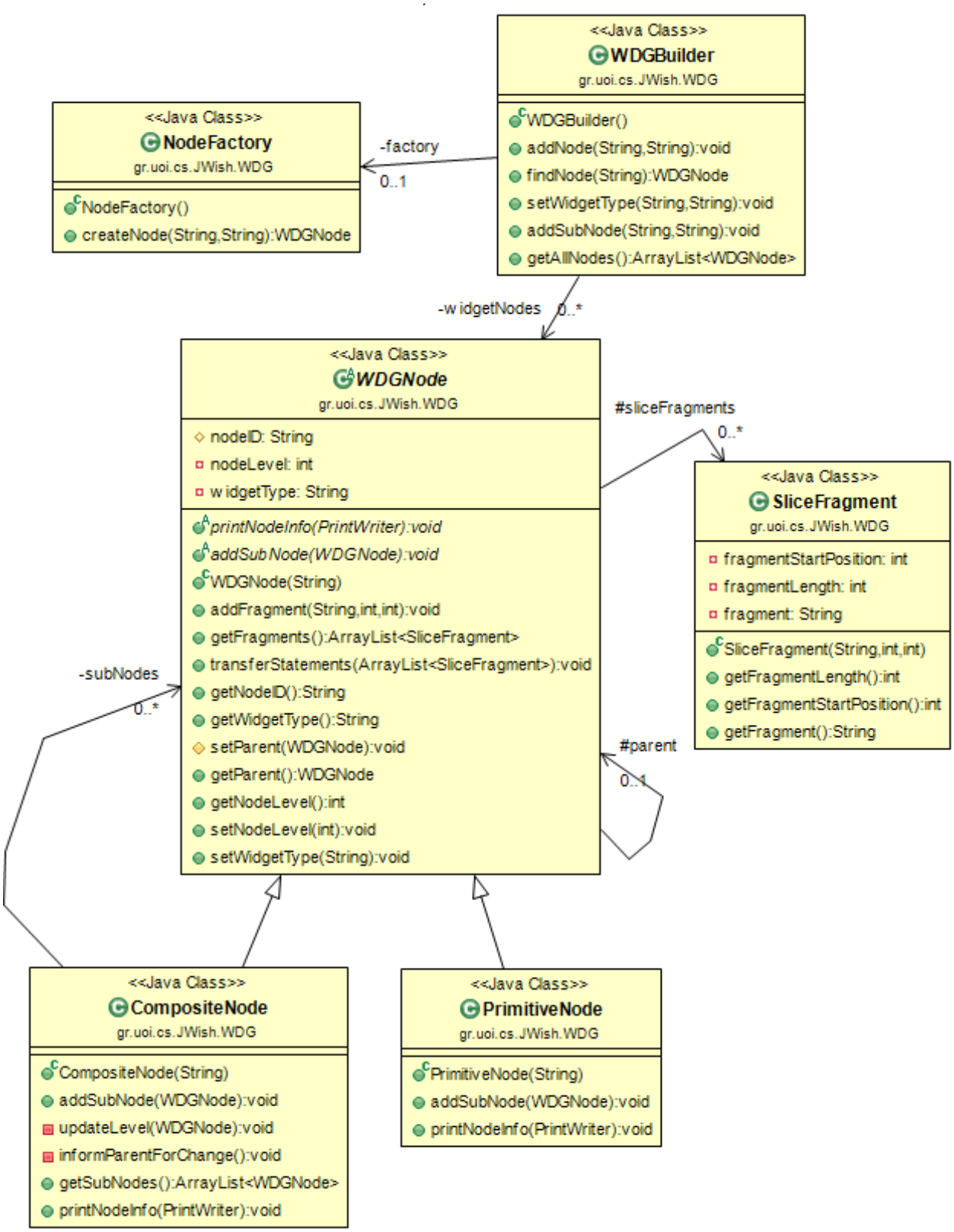

Figure 4.4 – UML Diagram of WDG package.

Slicer: In figure 4.5 we can see the UML diagram of the "Slicer" class. This is the class that finds widget slices from the WDG, given the slicing criterion. We use different methods for each criterion. In Table 4.1 we can see the match between criteria and corresponding methods.

Criterion	Matching method
<widget-name>	getWidget(widget-name)
<level>	getAllWidgetsOfLevel(level)
<widget-type>	getAllWidgetsOfType(widget-type)

Table 4.1 – Matching criteria and methods form "Slicer"

By setting level to 0 and using "getAllWidgetsOfLevel(level)" we can access all the Primitive nodes of the graph.

Also, we can create new methods that combine criteria to get different widget slices. For example, a new method could combine the criteria <level, widget-type> to produce different widget slices, like method "getAllSpecificLevelAndTypeWidgets()". Therefore, if we want to extract in classes all the menus of level 2 (or >=2) it would be useful to separate them from other widgets that can be high level, like Panels.

In figure 4.6 we can see some widget slices, as they were generated by JWiSH from the WindowBuilder test case.

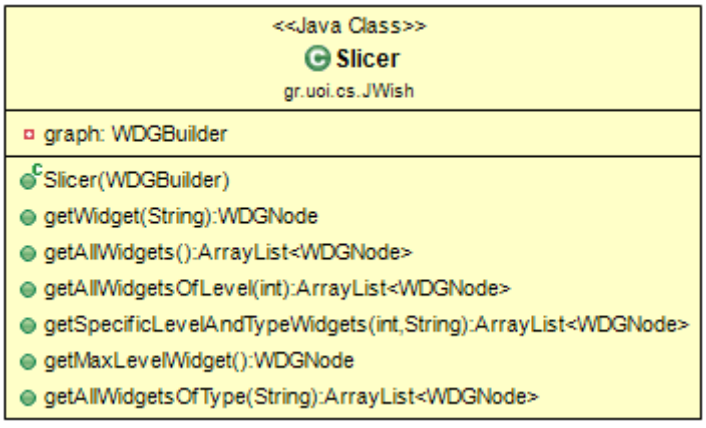

Figure 4.5 – UML diagram of Class "Slicer".

```
483 *************** SLICE OF mntmCut*************
484 JMenuItem mntmCut=new JMenuItem("Cut");
485
486 mntmCut.addActionListener(new ActionListener(){
487   public void actionPerformed(  ActionEvent e){
488   }
489 }
490 );
491
492 mntmCut.setEnabled(false);
493
494 mntmCut.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/cut_16.png")));
495
496
497 *************** SLICE OF mntmCopy*************
498 JMenuItem mntmCopy=new JMenuItem("Copy");
499
500 mntmCopy.addActionListener(new ActionListener(){
501   public void actionPerformed(  ActionEvent e){
502   }
503 }
504 );
505
506 mntmCopy.setEnabled(false);
507
508 mntmCopy.setIcon(new ImageIcon(EclipseGUI.class.getResource("/images/copy_16.png")));
509
510
511 *************** SLICE OF mntmCopyQualifiedName*************
512 JMenuItem mntmCopyQualifiedName=new JMenuItem("Copy Qualified Name");
513
514 mntmCopyQualifiedName.addActionListener(new ActionListener(){
515   public void actionPerformed(  ActionEvent e){
```

Figure 5.1 – Widget Slices generated by JWiSH

5. Validation

In this chapter we are going to test the functionality of *JWiSH* and present our results. For our experiments, we used as test cases the source code generated by four popular GUI-Builders. These GUI-Builders where used by Mamalis [Mamalis 2017] to create replicas of the basic structure of the Eclipse IDE, as mentioned in Chapter 2. The four test cases are presented in Table 5.1 below:

Case Study	**Widget #**	**LOC**
WindowBuilder	575	4316
NetBeansEditor	681	8141
JFormDesinger	573	6587
SWT	762	5390

Table 5.1 – The four test cases.

We are going run different experiments for each test case and present our results in detail. For validity purposes, it's important to mention that, although the first three test cases use Swing to generate the source code, the approach and style in the implementation of GUI is different.

Experiment 1: In this experiment we tested the time it took for the construction of the WDG in each case. The time displayed is the average after running *JWiSH* 5 times for each case study. Table 5.2 shows our results. Then we tried to determine if there was a connection between size and construction time.

Case Study	Average Time(seconds)
WindowBuilder	0,86342
NetBeansEditor	0,979033
JFormDesinger	0,901929
SWT	0,898801

Table 5.2 – Average WDG construction time.

To find the relation between size and time we used the Spearman correlation that measures the statistical dependence between the rankings of two variables. It ranges from [-1, 1] and the closer it is to 0 the more independent the two variables are (closer to -1 means negative correlation and closer to 1 positive correlation). We compared the average time with the number of widgets and then the average time with the lines of code. In table 5.3 we can see the results.

Compared Variables	Spearman Correlation
Time – widget #	0
Time - LOC	1

Table 5.3 – Correlation of size and time.

So, we concluded that the construction time of the WDG is independent from the number of widgets in the GUI, but it directly relates to the lines of code a GUI file has.

Experiment 2: In this experiment we took a sample of 25 Primitive Widget Slices and 10 Composite Widget Slices (regardless of level) from each test case, to see if their contents were correct. We used criterion <widget-name> for this experiment. To get an idea of the sample size observe in table 5.4 the number of Primitive and Composite Slices in each test case and then the results of this test in table 5.5.

Case Study	Primitive Slices #	Composite Slices #
WindowBuilder	500	75
NetBeansEditor	614	67
JFormDesinger	500	73
SWT	582	180

Table 5.4 – Number of Primitive and Composite Slices.

Case Study	Correct Primitive Slices	Correct Composite Slices #
WindowBuilder	25/25	10/10
NetBeansEditor	25/25	10/10
JFormDesinger	25/25	10/10
SWT	25/25	10/10

Table 5.5 – Correctness of Primitive and composite Widget Slices.

The conclusion from this experiment was that the construction and contents of widget slices are precise and accurate.

Experiment 3: In this experiment we tested the time required for slicing the WDG with criterion <widget-name>. In figure 5.1 we present a more detailed distribution.

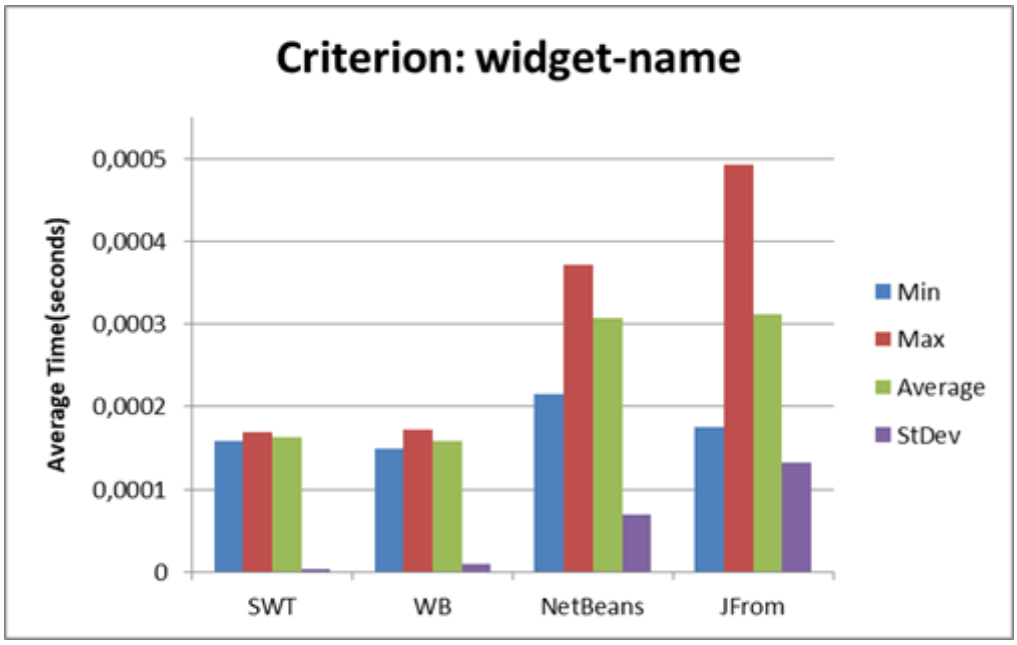

Figure 5.1 – Time distributions for slicing with criterion <widget-name>.

The sample used was the same as in Experiment 2. We concluded that the slicing time doesn't relate to the total number of widget slices, as SWT has by far the most.

Experiment 4: In this experiment we used criterion <level> to construct Composite Widget Slices and checked if they were correct and the time it took for slicing the graph. We used 0 and 1 for levels. In table 5.6 we can see the number of widgets we found for each level. In figure 5.2 and 5.3 we can see a detailed time distribution for levels 0 and 1 respectively.

Case Study	Level 0 widgets #	Level 1 widgets #
WindowBuilder	500	54
NetBeansEditor	614	51
JFormDesinger	500	52
SWT	582	85

Table 5.6 – Number of widgets for each level.

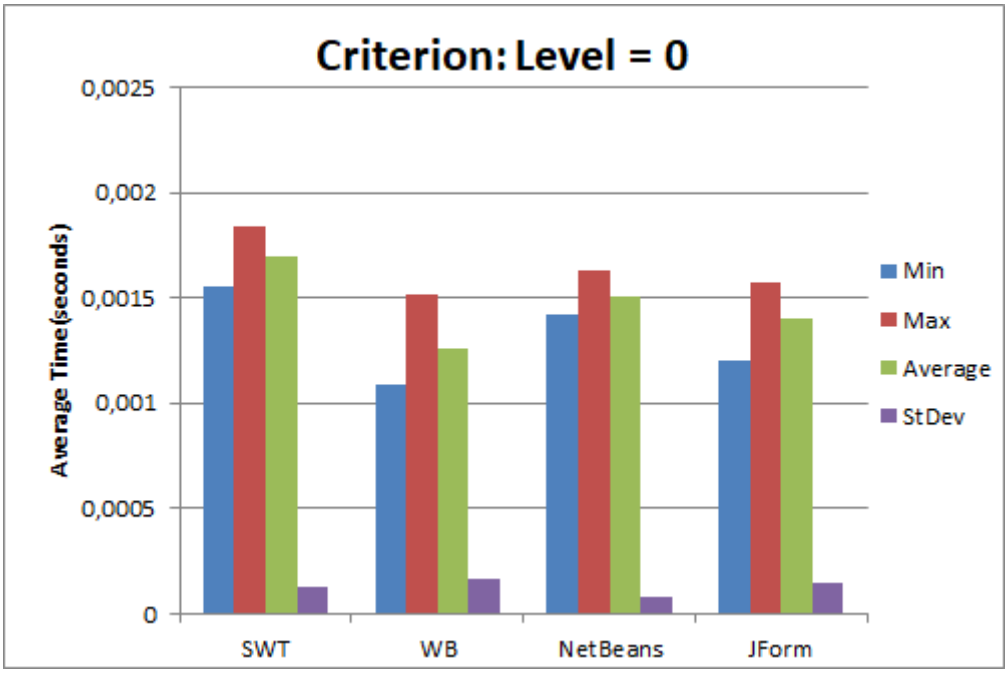

Figure 5.2 - Time distributions for slicing with criterion <0>.

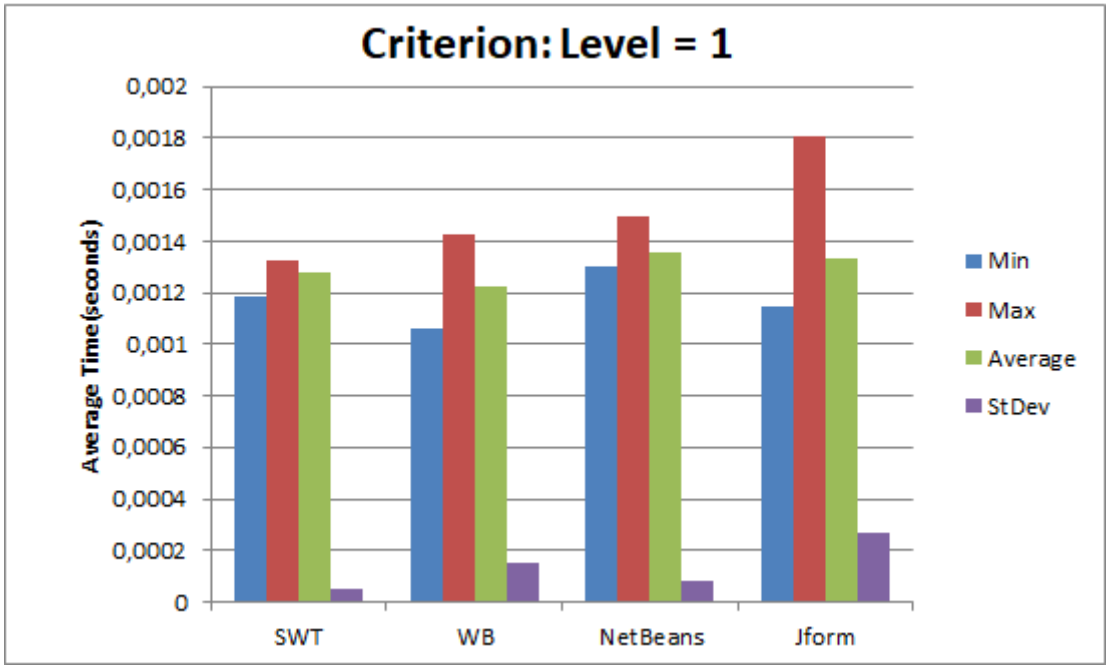

Figure 5.3 - Time distributions for slicing with criterion <1>.

The slices where accurate and here too, we concluded that the slicing time doesn't relate to the total number of widget slices or the level selected, as we can see in figure 5.4 below.

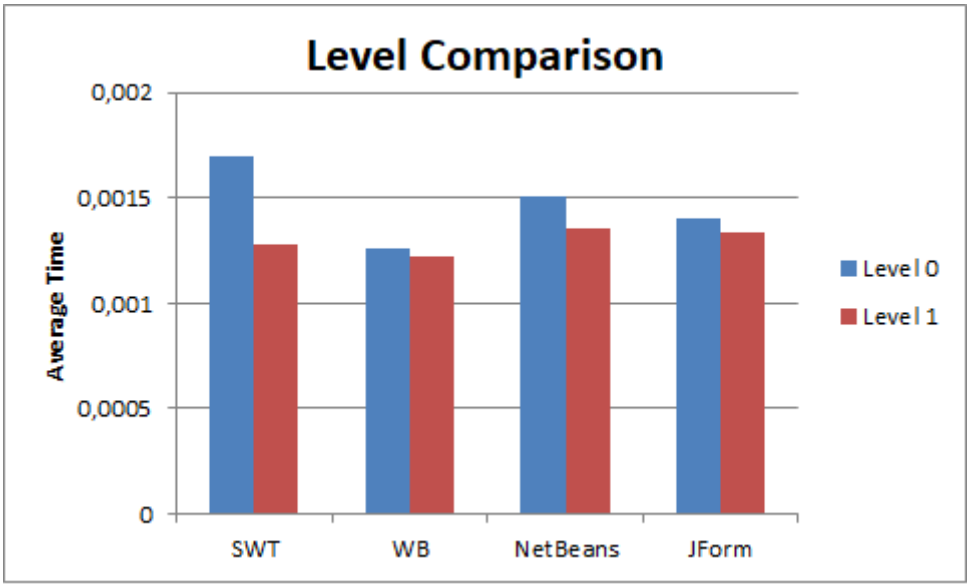

Figure 5.4 – Time comparison of levels 0 and 1.

Experiment 5: In this experiment we used criterion <widget-type> to construct again Composite Widget Slices and checked if they were correct and the time it took for slicing the graph. We used *Menu Items* and *Buttons* for the widget type. In table 5.7 we can see the number of widgets we found for each type. In figure 5.5 and 5.6 we can see a detailed time distribution for types Menu Item and Button respectively.

Case Study	Menu Items #	Button #
WindowBuilder	470	16
NetBeansEditor	448	20
JFormDesinger	470	16
SWT	439	11

Table 5.7 - Number of widgets for each type.

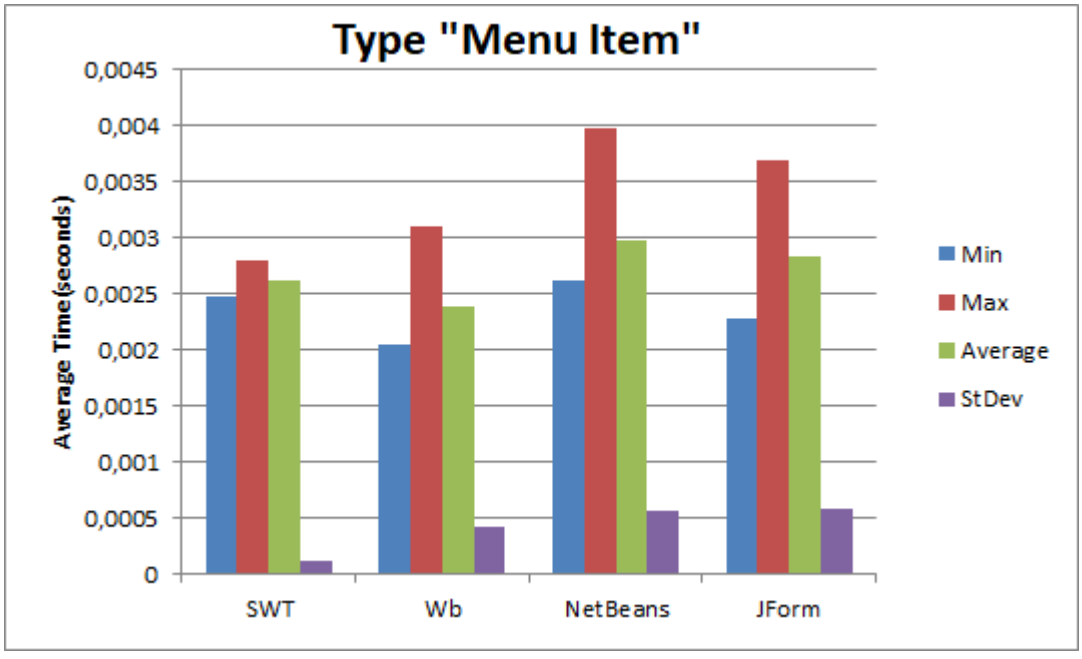

Figure 5.5 - Time distributions for slicing with criterion <Menu Item>.

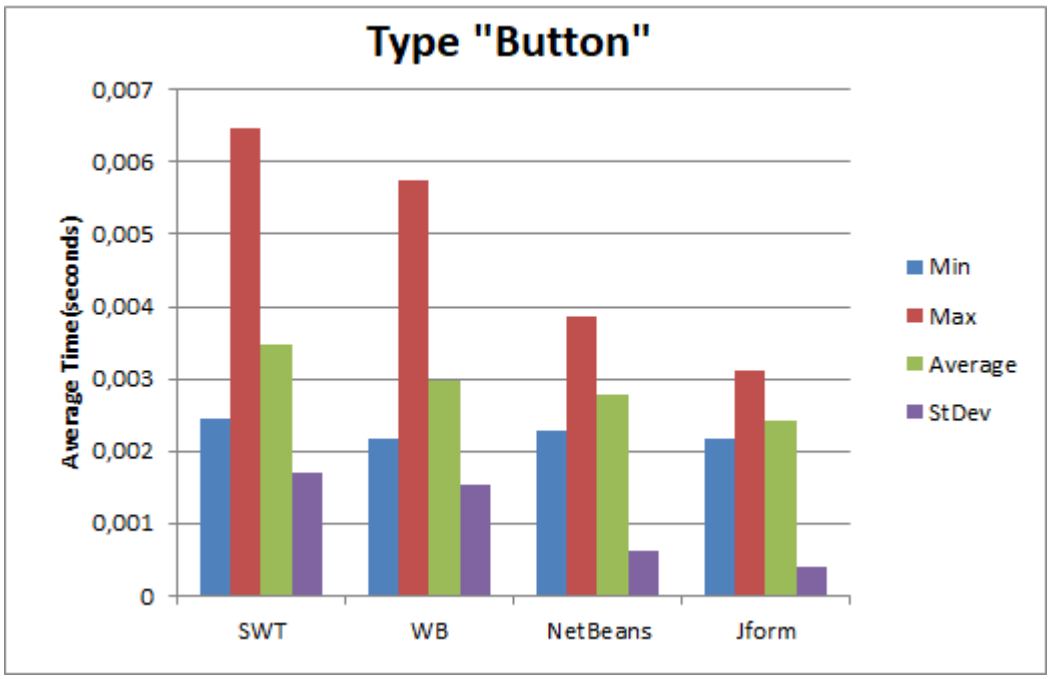

Figure 5.6 - Time distributions for slicing with criterion <Button>.

The slices where accurate and we concluded that the slicing time doesn't relate to the total number of widget slices or the number of the specific type widgets.

Experiment 6: In this experiment we tested the performance of JWiSH in an artificial, bigger GUI specification. We put WindowBuilder, NetBeansEditor and JFormDesinger together to create a more verisimilar GUI that consists of **3** files, **1829** widgets and **19.044** lines of code. The average time, after running JWiSH 5 times for this test case is **1.23** seconds.

After these experiments we can see that JWiSH is able of detecting and constructing widget slices quickly. It correctly identifies the level of a widget, the statements that affect it and its dependence on other widgets. The creation of the Widget Dependence Graph is rather quick and the slicing of it, using the different criteria, is accurate.

6. Conclusion

6.1 Summary

The purpose of this Thesis was to propose a technique that gathers all the information necessary, concerning widgets, for the automation of refactoring techniques applied to long and complex GUIs.

We created a new kind of program slicing specifically for GUIs. We named our technique *Widget Slicing* and analyzed the theoretical aspects and properties of widget slices. Also, to help construct widget slices, we proposed a new dependence graph especially for GUI elements called Widget Dependence Graph. Then, we developed an elegant tool that finds these slices, named JWiSH. We analyzed the basic architecture and design of it, as well as highlighted key points for using and extending it. Finally, we presented a variety of experiments using source code generated by popular GUI-Builders to test the functionality of JWiSH and its accuracy on finding widget slices.

6.2 Future Work

Possible future work considering Widget Slices could be:

- Further enrich JWiSH by adding new Parsers to help expand in other programing languages other than Java, or new APIs to increase its capabilities of widget slicing even more GUI building tools.
- A technique for analyzing widget slices and then using them for automatically applying different refactoring techniques to GUIs. Possible refactoring techniques activated by widget slices could be to extract the Composite Widget Slice of a high level widget in a class for a finer structure or create widget initialization methods. For example, you could determine which same methods are invoked by Buttons and engulf them in a new method (figure 2.4).

References

[De Lucia et al. 2003] De Lucia, M. Harman, R.M. Hierons and J. Krinke 2003. Unions of slices are not slices. In proceedings of the 7th European Conference on Software Maintenance and Reengineering, 26-28 March 2003, Benevento, Italy, IEEE Computer Society, 363-367.

[GoF 1994] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994) Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 97 ff. ISBN 0-201-63361-2.

[Josep Silva 2012] Josep Silva 2012. A vocabulary of program slicing-based techniques. ACM Computing Surveys (CSUR) Volume 44, Issue 3, June 2012, Article No. 12.

[Kollias 2018] P. Kollias 2018. Anti-Patterns in Code that is Generated by GUI Builders.

[Mamalis 2017] G. Mamalis 2017. Evaluation of the quality of the source code generated by GUI Builders for Java.

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

[Petridis 2018] Christos Petridis. JWiSH. Available at https://github.com/chpetridis

[Weiser 1984] Mark Weiser 1984. Program Slicing. IEEE Transactions on Software Engineering 10, 4, 352-357.