View-Based Maintenance of Graphical User Interfaces

Peng Li
Software Practices Lab
University of British Columbia
lipeng@cs.ubc.ca

Eric Wohlstadter
Software Practices Lab
University of British Columbia
wohlstad@cs.ubc.ca

Abstract

One difficulty in software maintenance is that the relationship between observed program behavior and source code is not always clear. In this paper we are concerned specifically with the maintenance of graphical user interfaces (GUIs). User interface code can crosscut the decomposition of applications making GUIs hard to maintain. A popular approach to develop and maintain GUIs is to use "What you see is what you get" editors. They allow developers to work directly with a graphical design view instead of scattered source elements. Unfortunately GUI editors are limited by their ability to statically reconstruct dynamic collaborations between objects. In this paper we investigate the combination of a hybrid dynamic and static approach to allow for view-based maintenance of GUIs. Dynamic analysis reconstructs object relationships, providing a concrete context in which maintenance can be performed. Static checking restricts that only changes in the design view which can meaningfully be translated back to source are allowed. We implemented a prototype IDE plug-in and evaluate our approach by applying it to five open source projects.

1. Introduction

In this paper we investigate a hybrid dynamic analysis and static checking approach for applying "What you see is what you get" (WYSIWYG) graphical user interface (GUI) editors to a dynamically rendered GUI view. We start by discussing the problem domain of GUI maintenance, existing tools for GUI development, our research focus and proposed contribution.

One difficulty in software development is that the relationship between observed program behavior and source code is not always clear [11, 36]. Typical software maintenance tasks require that programmers verify program correctness by examining program output (e.g. textual, graphical and message passing). Examination of output might be done manually or through the help of automated test cases [22, 23, 37]. When some problem is detected in the output, a typical strategy would be to start examining the code at the point where the problematic output was generated.

AOSD'08 March 31 -- April 4, 2008, Brussels, Belgium

In some cases it could be cumbersome for programmers to map output back to the statements which generated the output. This is because output generating statements can crosscut the implementation of several source modules. For example, typical logging frameworks and language pre-processors provide support to include both the name of the class and a line number where a logging statement was generated; so developers can avoid this problem.

In this paper we are concerned specifically with maintenance of GUIs. Our discussion focuses on one kind of program output, the rendered on-screen interface. We refer to the dynamically rendered, runtime, user interface as a *dynamic view*. Graphical user interfaces are pervasive in today's software systems and can constitute as much as half of the source code in typical projects, according to a published study [26].

A dynamic view can be interpreted by the programmer as one conceptual module (i.e. concern [33, 34]) from the perspective of a software maintenance task. We say conceptual because the dynamic view may not map directly to one source code module. Still, by looking at a dynamic view, the programmer may see a cohesive set of elements (called widgets), with a clear set of relationships. Relationships are formed from a hierarchical (tree-based), component-oriented model [6, 20, 32]. A dynamic view provides a complete, dynamic context, from which to understand specific containment relationships. Mapping all these widgets and the relationships to source, however, requires understanding a scattered set of source elements. Just as all logging output would not be confined to be generated from a single module, neither is a dynamic view confined to a single module. We seek to understand possible solutions to this software maintenance problem. A possible solution might be found in existing tools for development of GUIs.

1.1 GUI Editors

A popular approach to develop GUIs is to use WYSIWYG editors, henceforth called *GUI editors* [15, 30, 31, 35]. GUI editors use a static view of the GUI, called a *design view*, to support a developer in two ways. First, by selecting widgets in a design view, the editor can display where in the source code the widget was created or some property of the widget was modified. This is useful when a developer wants to make some changes to the GUI manually in the source code. This kind of hyper-linking between output and source is conceptually similar to that of the simple logging example. We call this practice *view-based navigation.*

Second, by manipulating widgets in the view, a developer can affect changes on the GUI. We call this practice *view-based editing*. These changes are translated by an editor into changes on the underlying source code. This allows the developer to reason vi-

sually about graphical changes. Changes which are made from a design view are limited to "design-time" changes which are static in nature; since the visual nature of the editor lends itself to representing graphical relationships and not computations. Together we refer to the use of view-navigation and view-editing during software maintenance tasks as *view-based maintenance.*

Typical GUI editors provide a round-trip (re-) engineering [27] process. A design view created in the editor, can be used to generate source. Later, the source code can statically be reverse-engineered to recover the design view.

1.2 Research and Contributions

Unfortunately, the reverse-engineering capabilities of GUI editors are severely limited. This is because complete applications consist of both the user interface design and dynamic collaborations between objects which implement the behavior of the program. These two parts of the program work together to provide a complete system and can be tangled together in the program implementation. Ensuring a complete separation of the user interface design code and the program behavior would require significant effort from developers. Developers may also have prioritized decomposition decisions along other dimensions of concern which conflict with this separation [34]. In 4 out of 5 open source applications that we looked at, there was not sufficient separation to allow a commercial off-the-shelf editor (COTS editor) [31] to understand the user interface. We quantify these results as part of our evaluation.

The intuition behind the approach is based on an observation that much of the information in a dynamic view (as in Figure 1) consists of component sub-tree clusters that are static in nature: i.e. making heavy use of Singleton [8] classes and constant expressions. However, in practice, these sub-trees are often "glued" together with small pieces of dynamic computation. The approach used by traditional GUI editors cannot make sense of the dynamic code enough to find relationships between the static sub-trees, i.e. the analysis "gets stuck in the glue". In the end, the entire tree-based view becomes *truncated* (as in Figure 2). Sub-trees are no longer visible in context because they are not reachable from the root widget in a design view.

Figure 1: Screenshot from Java-Chess program execution. 16 classes contribute to this particular dynamic view.

In this paper we investigate a hybrid dynamic and static approach for improving view-based maintenance. Rather than use a static approach to recover the design, we use a dynamic approach. This provides a complete, dynamic context, for making informed maintenance decisions. However, this complicates translating view-edits back to source, since part of the view is now the result of dynamic computation. So, we provide constraints based on static checking to limit what can be edited. Our contribution is to describe this hybrid approach and show for a case study of five open source applications that it can be used to:

- Provide view-based navigation on a complete dynamic view.
- Increase the amount of static design information available for WYSIWYG-based editing.

We describe a prototype tool which supports both view-based navigation (Section 4) and view-based editing (Section 5) in a dynamic context.

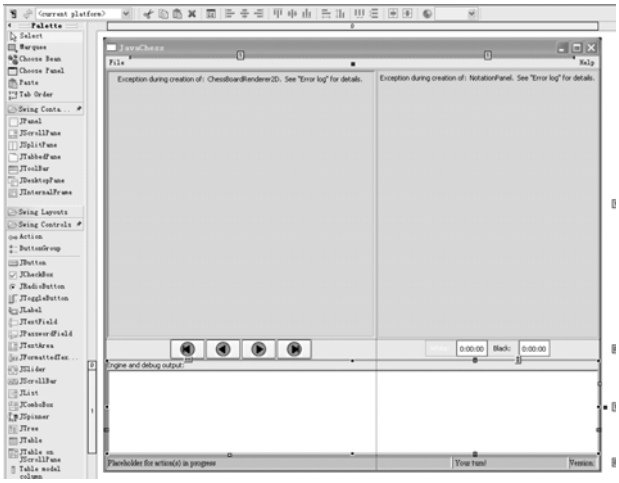

Figure 2: Screenshot from Swing Designer GUI editor for Java-Chess class. Several widgets are missing from the recovered design view. Extra rulers and grid markers are the effect of the editor tool in action.

The rest of the paper is structured as follows: Section 2 presents a concrete motivating example, Section 3 presents background on GUI editors, Section 4 presents view-based navigation, Section 5 presents view-based editing, Section 6 presents a quantitative evaluation, Section 7 presents a discussion and limitations, Section 8 presents related work and we conclude in Section 9.

2. Example with Java-Chess

Consider a scenario where a developer wants to perform maintenance on the user interface for a chess game, in particular the open source Java-Chess [14]. A developer may first need to plan a change they intended to make, by reasoning about some dynamic view, as shown in Figure 1. Dynamic views are a product of program execution so they are not directly available for editing by the

Figure 3: Screenshot from Swing Designer GUI editor for Java-Chess augmented with our dynamic analysis. Sub-trees lost in Figure 2 have been "glued" back on. Extra rulers and grid markers compared to Figure 1 are the effect of the editor tool in action.

developer. So, when a programmer has an intended change in mind, they would need to consider how to implement the change. Two possibilities for implementation are to make changes to the source directly or use the help of a GUI editor: we consider each of these in turn.

First, to change the source directly, the developer would need to know the method call source element which generated the widget (or widget property) that they want to change. These calls could take some time to find since the dynamic view may crosscut the source. In Figure 1 there are 16 classes which contribute to the visual appearance that include GUI code tangled with program logic. A developer familiar with the code base would need to remember how the statements of code in all these classes related to the dynamic view or else reconstruct all these relationships.

A second option for the developer would be to use a GUI editor. It would seem reasonable that the developer should be able to simply click on some part of the view and ask to be taken to the source element where that part of the view was generated. Also, for parts of the view that are static design, the developer should be able to edit directly from the view. This is possible in very limited cases with existing GUI editors as we discuss next.

Current GUI editors are crippled once the initial GUI design code has been integrated with the main program logic. As an example, consider the same view rendered by a COTS GUI editor in Figure 2. We can see that several of the widgets are missing including: two menubar menus, the entire chessboard, and the game player information panel (in the upper right quadrant). The particular GUI editor to produce Figure 2, Swing Designer [31], was rated as the most complete Eclipse-based editor by an online article [30]. This is unfortunate considering that much of the information missing is part of the static design. As an example, the missing menu items and associated drop-down menus never change during the course of Java-Chess execution.

This scenario is used to demonstrate that mapping intended changes from the dynamic view to the implementation is difficult to do manually and not possible for complete programs using existing editors.

We seek to address the problems by allowing developers to interact directly with a dynamic view even for complete programs. The developer executes the program they want to edit. A dynamic analysis creates a mapping between the dynamic view and the source code, and static checking prevents changes to elements which are not part of the static design. Figure 3, shows the Java-Chess view again, using the same COTS editor augmented with our analysis. The view now provides a complete dynamic context for design maintenance.

3. Background

Object-oriented GUIs make use of objects called widgets [22]. First, we describe the unique characteristics of these components (Section 3.1). Then, we describe the process for engineering user interfaces using a traditional GUI editor, as in Figure 4. The steps for forward engineering an interface from a high-level design are shown from left to right (Section 3.2). Then, from right to left we see the steps for reverse engineering a design view from source code (Section 3.3).

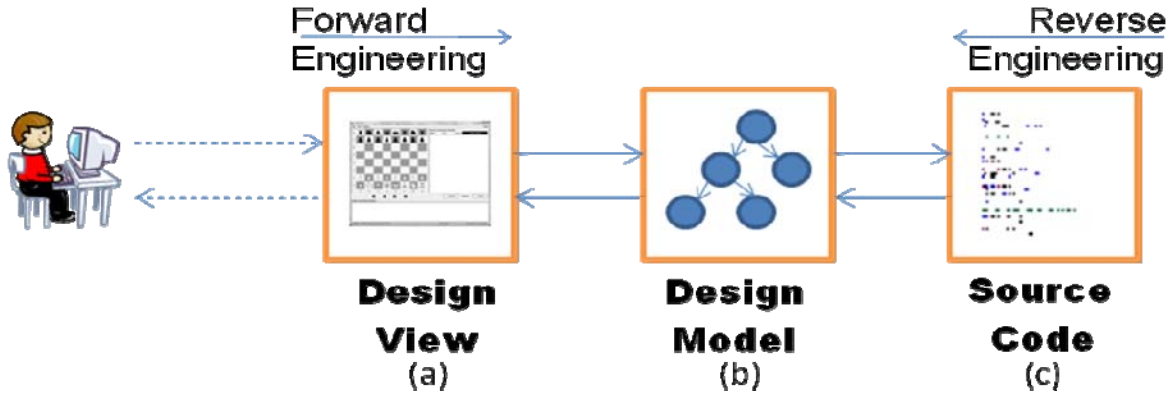

Figure 4: Forward and reverse engineering process for graphical user interface editors.

3.1 Widgets

A *widget* is an object created from a specific set of classes provided by a GUI framework or a custom, application specific, subclass of those classes. We call these classes *widget classes*. GUI editors, much like component-based programming and dynamic languages (e.g. Smalltalk [17]), often blur the lines between compile-time and runtime, so we are careful to use the word widget referring to an object and widget class referring to a class.

The state of a widget consists of: *widget properties* and possibly other *child widgets*. Throughout the paper we refer to both *design* and *dynamic* models of widgets. A design model is one which is constructed using only compile-time GUI design information, whereas a dynamic model can only be observed at runtime (i.e. a heap "snapshot").

The distinction between objects constructed directly from *framework classes* (e.g. Frame, Button, Window) and objects constructed from *custom classes* is important. A GUI editor understands the semantics of framework widgets without any analysis of framework code. Knowledge of framework widgets is hardwired into the editor itself. Essentially, during static design model recovery, the editor is able to treat framework widgets as if they were program constants (i.e. string literals, number literals, etc...). This also includes the use of specific localization resources for storing and retrieving strings in different languages. We refer to the union of these framework widgets and actual program constants as *GUI constants*. For example, a call to "`new Frame()`" is considered a statically resolvable constant, so interpreting this statement allows the editor to include a Frame widget in the static design model.

3.2 Forward Engineering

Forward engineering begins with the developer who envisions the design of the graphical user interface. A developer works directly with the design view of an editor by manipulating widgets (a). As changes are made to the design view, the editor maintains an internal hierarchical data-structure (i.e. object graph) of widgets, the design model (b). Each change to the design model can either: add a new widget somewhere in the hierarchical model, remove a widget, or change a property of a widget. Finally, the editor realizes the implementation of the GUI by generating code from the corresponding design model (c).

3.3 Reverse Engineering

GUI editors provide reverse-engineering by allowing a user to select a custom widget class (c) that they want to edit. For example, to provide Figure 2, we chose the custom class named "JavaChess" from the Java-Chess project. The editor analyzes the constructor of the class and recovers a design model (b) for an instance of this root widget to display in the view (a).

3.3.1 Design Model Recovery

During this reverse-engineering process, design models are statically recovered using information in the constructors of widget classes[1]. This is where a large part of the complexity of GUI editors lies, so we describe the details.

First, any statements which use variables that cannot statically be determined to refer to a single GUI-constant, are removed from consideration. This determination is limited to the analysis of the constructor and any fields which include initializers[2]. Therefore traditional compiler (intra-procedural) analysis is sufficient to make a conservative approximation. Next, any statement nested in loops and conditionals[3] is removed from the consideration, including the loop or conditional. Now, the only code left, is a sequential list of statements only making use of GUI-constants. This code can now be interpreted by the editor to build the statically determined model.

3.3.2 Model Synchronization

The important result of this entire process is that a one-to-one correspondence between source statements and the widgets in the design model of a custom class is created. So changes to either can be synchronized [27]. This is because only functions that are a one-to-one correspondence have a well specified inverse [21]. Since the code of framework (library) widgets cannot be modified, editors translate changes on framework widgets to the source of the closest custom class (least ancestor) to which they are children. This custom class provides the scope for changes to a framework widget. We call such a class, the *change context*.

Framework widgets from the same class, inside this context, can be distinguished by the editor. For example, if a custom Panel contains two Buttons, then changing the color of one Button will not affect the color of the other Button. Contrary to this behavior, all editors, to our knowledge, do not distinguish between different instances of custom widgets based on their context. For example, any changes to some GamePanel custom widget will change the source of the GamePanel class and therefore affect all GamePanel instances. For better or worse, we have not tried to improve that particular property of GUI editors so we also inherit this restriction (discussed further in Section 7).

Dependence on any dynamic information can result in only a partial view being constructed (as in Figure 2). This can make it hard for the developer to make any changes to design information contained inside some particular dynamic view.

4. View-Based Navigation

View-based navigation provides hyper-linking between the GUI editor and source code locations. First we explain this relatively straightforward "read-only" support of our tool. Then in Section 5, we build on these details to explain the more complicated "read/write" editing support of our tool.

Our approach makes use of information derived from the dynamic analysis of program execution. Our Eclipse IDE plugin creates a dynamic model of the GUI, effectively replacing the design model normally used by the editor (as in Figure 4, b).

For our tool to create a dynamic model of some GUIs, a developer is required to execute the program; making sure a dynamic

[1] As well as private methods called during constructor execution.

[2] This excludes public static fields which could have been set previous to constructor execution.

[3] This also includes any code which might execute after a return statement controlled by a conditional, and recursive methods. We do not consider exceptional control-flow or concurrency in our implementation.

view the developer is interested in is rendered during that execution. After this analysis, the developer can navigate between widgets in the dynamic view and source code elements using the plugin. This provides a complete, dynamic context, for view-based navigation.

During program execution, interception of method call join points specific to the GUI concern is used to create a dynamic model. We use AspectJ to collect this information about the GUI. We created an aspect, `GUIAspect`, to monitor three important kinds of events related to the user interface. For our prototype, applications using the Java Swing API are supported. Below we provide simplified pointcuts from our implementation to illustrate.

First, the creation of widget classes is intercepted according to this construction pointcut:

```
call(javax.swing.*+.new(..))
```

Second, we need information regarding any changes to the properties of a widget. This is done through the pointcut:

```
call(void javax.swing.*.set*(..))
```

Finally, we need information regarding the containment relationship between container widgets and their child widgets. This is achieved through:

```
call(* javax.swing.*.add(..))
```

In each of the advice, the reflective capabilities of AspectJ are used to record information for argument values and corresponding source locations (i.e. classes and line numbers). This information is used to create a map between widgets and source code, for supporting view-based navigation.

Once the user is satisfied that the view they are interested in has been displayed during execution, the user can select an option from our plugin, which will in turn provide the currently constructed dynamic model to a COTS GUI editor. The actual underlying data in the design model is now replaced with a dynamic model. We used this approach simply to avoid making a custom GUI editor. Now when the user selects widgets in the GUI design view, they can choose to be hyper-linked to the source code statement of the corresponding GUI joinpoint as captured by `GUIAspect`.

In our evaluation (Section 6), we quantify the number of classes that contribute to the generation of a typical dynamic view for open source applications. Since source locations across all these classes are now made available to a developer from a single high-level conceptual module (the view), this metric helps provide some measure of the reduction in information that must be reasoned about by the developer for navigation.

4.1 Custom Graphics

In many user interfaces, part of the dynamic view may consist of custom animations or graphics generated dynamically using calls to a 2D graphics package. These graphics do not consist of widgets but rather are translated directly to a two-dimensional array of pixel elements to be displayed on screen. In Swing, custom graphics are implemented using objects that support a "callback":

```
public void paint(Graphics g);
```

The implementation of the method makes use of the `Graphics` object to draw custom graphics.

`GUIAspect` advises this method using an `after` advice and captures the `Graphics` argument. After `paint` has executed, the body of the method will have written information to the object. We extract this information and save it on disk as a graphics file. Then we place an `Image` widget in the model as a "proxy" for the custom graphics object, so this image is displayed when the dynamic model is rendered by the editor.

During view editing (Section 5), this provides the developer with a "snapshot" of the animation or graphic. This is useful for providing a dynamic context to inform maintenance decisions. For example, in the Java-Chess application (Figure 1) the entire chessboard is drawn using graphical rectangles. The chessboard squares are not widgets; but the chess pieces are widgets. So the chessboard cannot be edited from a design view but the chess pieces could. However, without the dynamic context of the chessboard, it would be difficult for a developer to make an informed design decision about potential changes to the chess pieces. So we see how even "read-only" information can be useful to provide a dynamic context for view-editing, which we describe next.

5. View-Based Editing

First, we discuss the technical difficulties of view-based editing in the context of a concrete code example to further explain why some constraints are needed. Then we present our approach.

Figure 5 shows the Java-Chess source code for generating the chessboard column labels seen on the bottom of the chess board in Figure 1. These labels indicate the columns `a-g`. This example demonstrates a situation where a one-to-one correspondence between source and a design model cannot be created. In this case we must prevent editing from the design view.

```
1. ...
2. for(int i = 0; i < 8; i++) {
3.   char column = 'a' + i;
4.   JLabel label = new JLabel();
5.   label.setText(column+"");
6.   this.add(label);
7. }
```

Figure 5: Java-Chess source code in the constructor of chess board column panel. Code has been refactored to improve readability.

In the source code, we see that column labels are generated by a loop (lines 2-7). The characters to be displayed are created using the loop counter and adding it to the program constant `'a'` using arithmetic addition of character codes (line 3). Then each label widget is created and its text property is set (line 4-5). Finally, each label is added to the panel (i.e. this) on line 6.

It is very difficult (or even impossible) to translate a change in one of the characters `a-g` from an editor design view to some underlying change on the source, without completely disrupting the source structure.

Suppose a developer changed the label ‘g’ to the label ‘G’ in the editor design view with our dynamic model installed. If the editor naively translated line 5 to `setText("G")`, we would end up with eight labels all the same!

Recall that a design model is simply a tree data-structure of widgets. Considering line 5, an editor could not create a one-to-one correspondence between this statement and a design model for two reasons. First, `label` refers to more than one object during the constructor execution. Second, `column` does not refer to a GUI constant. Even though our tool can display such computed information, we still need to respect the limitations of graphical-based editing.

5.1 Editing Constraints

We have seen that only changes on the design view which correspond one-to-one with source code changes can be allowed. However, since our tool has allowed the developer to view dynamic context, our tool needs to provide guidance to prevent editing of dynamic information. COTS GUI editors also must wrestle with this fundamental problem. Their approach is to not to display any dynamic information. They only display the information for which a one-to-one correspondence has been created. We have investigated the use of editing constraints.

We want to provide the same guarantees as traditional editors. Recall from Section 3.3.1 that only the design of a single custom class, the change context, is affected by edits. This allows developers to reason that the initial state of all widgets of that custom class will be affected in the same way by a change.

For example, consider that all chess piece widgets in JavaChess are created from the same custom class. Assuming that setting the size of a chess piece is part of its design, then changing the size of one chess piece will predictably change the initial size of all chess pieces. This is often what the developer would want, but not always. We chose these particular constraints to provide an equivalent behavior as existing GUI editors. Our implementation is an adaptation of the existing static design model recovery technique used by GUI editors (Section 3.3.1). Perhaps other less restrictive constraints could be applied, but we leave this to future work.

5.1.1 Constraint Details

When a developer makes a change to the design view with our plugin activated, that change is translated by the editor into a change to a widget. This widget will be part of the dynamic model collected by `GUIAspect`. A source code edit will be made by the editor to realize the change, if it is not prevented by our constraint.

First, we determine the potential change context. We determine the least ancestor of the affected widget that is a custom widget. In other words, starting from the widget in the component tree, we walk up the tree until we encounter a custom widget. Changes directly to custom widgets can be handled without this first step. Second, we determine the located custom widget’s class. This information is available because we have captured the model at runtime.

Next, the located custom class is analyzed using the traditional GUI editor static recovery technique. This gives us the static design model of the class. Finally, we can simply compare the subtree of the dynamic model rooted at the change context and the design model of its class. We need to determine if they “intersect” at the widget where editing is considered. This intersection is implemented using simple top-down tree-to-tree comparison [28]. If they do agree then the change is allowed, otherwise the change is prevented and a warning is issued. In the future we could provide a visual cue such as a highlighting or overlay which points out all of the static design information embedded in some dynamic view.

When classes contain multiple constructors, our implementation uses the intersection of static models recovered from all class constructors. This is implemented with top-down comparison as above. This is useful because we noticed that often multiple constructors are declared, which simply delegate construction of the GUI to a single private helper method.

5.2 Dynamic Context Improves View-Editing

A developer using our tool now sees both static design information and dynamic information in the editor view. Editing the design information will affect the initial state of all widgets from one custom class in the same way. Editing the dynamic information from the editor is not supported, although view-based navigation is still enabled. We saw in the previous section why some restrictions on editing dynamic information make sense.

Editing static design information is also supported by traditional GUI editors. So it may seem at first glance that the only thing we have contributed is view-based navigation for dynamic information. This is not the case. Here, we explain how our tool can help provide more context for editing design information.

To make this clear we provide a simple abstract example as shown in Figure 6. Here there are four widgets: frame, panel, label, and button. The edges which are solid are relationships derivable from a design model. The edges that are dotted are dynamic relationships, derivable only from a dynamic model. Source code creating these relationships is shown in Figure 7.

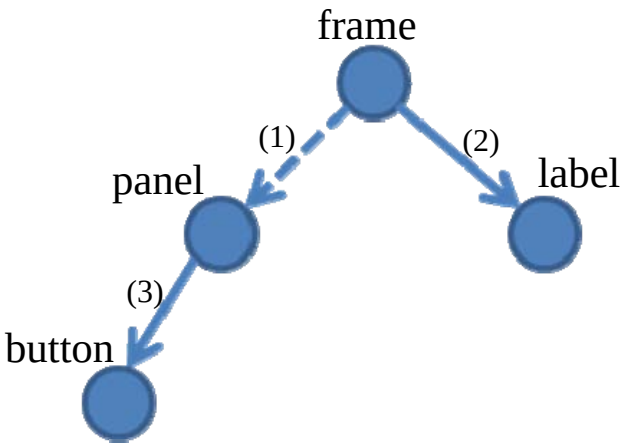

Figure 6: High level illustration of a dynamic model (object graph) including design (solid) and dynamic (dotted line) information. Numeric labels refer to “add” relationships between widgets in Figure 7. Design model will appear to the user as truncated version of dynamic model.

```
  /****Main.java****/
public static void main(String[] args) {
  MyFrame frame = new MyFrame(new MyPanel());
  ...
}

/****MyFrame.java****/
class MyFrame extends JFrame {
  MyFrame(JPanel panel) {
    add(panel);                    //(1)
    add(new JLabel(..));           //(2)
  }
}

/****MyPanel.java****/
class MyPanel extends JPanel {
  MyPanel() {
    ...
    add(new MyButton());           //(3)
  }
}

/****MyButton.java****/
class MyButton extends JButton {
      //Details elided
}
```

Figure 7: Three custom widgets classes used to demonstrate how dynamic context improves view-editing (an illustration is shown in Figure 6). Notice that since the argument in the MyFrame constructor requires knowledge of the dynamic type, it cannot be displayed in a traditional design view.

This scenario corresponds to the source code in Figure 7. When the frame is created in Main, it is passed a MyPanel as an actual argument. However since the JPanel formal argument of the MyFrame constructor could refer to any custom subclass of JPanel, a traditional editor would only display a single frame and a label (no panel would be displayed). Programs include many sources of information that can only be determined at runtime. Here we just use the dynamic type of an object as one example.

Using a traditional GUI editor, the frame could be viewed for editing because it is a custom widget. Although no panel or button design would ever be shown as contained inside the frame. A developer interested in maintaining a particular view of the user interface may find it frustrating that they are unable to edit seemingly static design portions of MyPanel in the visual context of the frame.

This demonstrates an important point from an aspect-oriented software development perspective. Hierarchal structures, such as the widget hierarchy, provide a disciplined way to organize modules so they easier to navigate. However, as we showed, the hierarchy displayed in existing editors can become truncated, leaving the user with a "flat" set of modules through which no relationships are provided.

A developer using our tool could execute the program under their control until some dynamic view was rendered that they needed to maintain. They might even control the program in such a way as to create relationships that would be helpful for them during the maintenance process. For example, if more than one kind of panel could be placed in the frame of Figure 7, they could choose to place the one they were interested in. They could perform view-editing or view-navigation in this visual context of their choosing. Certain dynamic information could not be edited from the design view. In Figure 7, this would be statements involving the relationship between panel and the frame. We provide some initial quantification of these potential advantages for specific applications and use-cases in the next section.

6. Quantitative Evaluation

Application Name	NCLOC
CrosswordSage	3,093
Java-Chess	5,616
jMSN	7,335
GanttProject	43,338
FreeMind	65,420

Table 1: Application subjects and their sizes in terms of non commented lines of code.

Here we provide a quantitative evaluation of our approach using five open source applications. First, for some specific program views, we want to measure how many classes contribute information to the view. This metric could support our claim that maintaining GUIs can be difficult and that view-based navigation is useful. Second, we wanted to measure how much of some view for each program could be reverse-engineered using a COTS GUI editor. This could support our claim that design code and program logic are often tangled, making design model recovery difficult. Third, we wanted to compare the amount of static design information available in a particular use-case, using the original and our proposed approach. This could support our claim that dynamic context provides more opportunities for design view-editing.

We initially looked at three existing Eclipse-based GUI editors for comparison with our approach: Swing Designer [31], Visual Editor [35], and Jigloo [15]. We chose Swing Designer to extend and compare against our approach. The extension is based on a standardized JavaBeans Customization API so we did not require any source code. We discovered from their documentation that all the editors use the same overall approach for static model recovery, described in Section 3.3.1.

Swing Designer and Jigloo performed consistently better than Visual Editor. They are both mature commercial products whereas Visual Editor is an emerging open source project. Swing Designer out-performed Jigloo so we only present metrics from Swing Designer. Our measurements from Jigloo are available online[4]. We

[4] http://purl.oclc.org/NET/gui.htm

have also found an online article that rates Swing Designer very highly [30] against other editors. These reasons make us confident that the tool we are comparing against is a fair representative of existing tools.

For the evaluation subjects, we selected five open source applications which are built on the standard Java Swing libraries. Our first subject, Java-Chess, was selected before our prototype was developed and was used to inform the creation of our approach. Four other subjects were selected after our prototype was developed. These programs were selected because they were used as subjects in a recent paper about testing GUI programs [37]. By using subjects selected by a third-party we hoped to provide objectivity in our results. Table 1 lists the names (column 1) and sizes (column 2) of each subject in terms of non-commented lines of code (NCLOC).

Since our approach is dynamic we needed to choose some time in the execution of each program for which to take measurements. We wanted this choice to be as unbiased as possible and to be as uniform as possible across all the programs. We noticed it was very common for Java applications with a GUI to include a default "main window" with a variety of panels, menubar, and toolbars. This main window and the widgets it contains are often loaded immediately upon execution of the program. Then, after the main window is rendered, the application becomes idle, waiting for user input. We felt this point in time where the application becomes idle made a good choice of time because it was easy to objectively determine this point in the execution of many different programs. We call the execution of a program up to this point, the "main window scenario".

6.1 Decomposition of main window view

Application Name	Classes Used
CrosswordSage	2
Java-Chess	16
jMSN	11
GanttProject	45
FreeMind	12

Table 2: Metric measures the number of classes containing GUI joinpoints captured by `GUIAspect` for rendering of the main window view.

This metric measures the number of classes containing method call joinpoints captured by GUIAspect for rendering the window of the main window scenario. This provides us with an indicator of the reduction of source classes that a developer must manually navigate when performing maintenance on this particular view of the program. Since classes are the primary unit of modularity in OO design, we believe a larger number of classes indicates more difficulty performing maintenance on the view without proper tool support. These results are displayed in Table 2. We believe that in at least 4 out of the 5 cases (where over 10 classes were used) there is evidence that view-based navigation would be useful.

6.2 View-Navigation

Application Name	Size of dynamic main window model	Size of original design model relative to dynamic model
CrosswordSage	143	100%
Java-Chess	291	45%
jMSN	149	34%
GanttProject	491	8%
FreeMind	109	10%

Table 3: Size metric is determined by the number of widgets, and set widget properties used in the dynamic model and the static design model.

Here we first measure the "size" of the dynamic model collected by `GUIAspect` and the size of the static design model recovered by the original GUI editor, for the main window view. This size metric was determined by the number of widgets and number of set widget properties used to render the view. In each case, it is the size of some tree data-structure in terms of these logical elements.

The actual dynamic model was determined by using GUIAspect and counting the size of the actual main window component graph through a traversal of objects. The models can seem quite large upon observation. We note that in addition to the information seen directly in the main window, the models include information about all menu items (and sub-menu items) in all drop-down menus of the window. Also, GanttProject is almost twice the size of the second largest model because the main window contains two "tabbed panes" with different user interfaces.

To record results for the COTS editor we needed to know the custom class which represented the top window for the main window scenario. This was easy to determine for all cases and was usually the class containing `main`. We input this class to the original COTS editor and recorded the numbers. These numbers are available from the tool itself. It is useful to note that all of the information displayed by standard editors is statically editable because they only display information recovered using static reasoning.

These results are displayed in Table 3. For one application, CrosswordSage, the complete main window view was recovered statically. However, we can see that the design model was less than 50% of the dynamic model in 4 out of 5 cases. So we can see that the opportunities for view-based navigation are limited in the current approach. Our approach captures the complete dynamic model; additional screen shots like the ones shown for Java-Chess are made available online[5]. This is to be expected since a dynamic analysis has complete runtime information. The only problem that

[5] http://purl.oclc.org/NET/gui.htm

would arise is if somehow the GUI was manipulated without being intercepted by `GUIAspect.` This was not a large problem in our case since Swing libraries strictly follow the conventions set out by a standardized component model (JavaBeans). We have carefully inspected the visual appearance for all cases. In each case there is less than three small visual differences which are caused by the fact that we have not yet implemented a pointcut for catching when widgets are disabled (i.e. "grayed out").

It is important to note that the COTS builder was not designed to work on arbitrarily tangled code bases. So the percentages in the table really tell us more about the structure of those code bases than about the tool itself. We feel the numbers provide some evidence that larger code bases tend to introduce more complexities. This could make the GUI hard to maintain without good tool support.

In some software development situations, smaller models might actually be better because they provide a useful abstraction. So, we clarify that in all these cases the design model is simply a truncated version of the dynamic model, cut off at some levels in the tree. The results in the table describe how much truncation occurs. The truncation is apparent visually for Java-Chess in Figure 2 and truncation is illustrated in Figure 6.

These measurements provide a basis for judging the usefulnesss of view-navigation on a dynamic view. They do not provide a good comparison for judging the view-editing capabilities of our approach, since much of the information in the dynamic model is not static design, so we explore this next.

6.3 View-Editing Comparison

Application Name	Original editor	Augmented Editor	Improvement
CrosswordSage	100%	100%	-
Java-Chess	45%	56%	1.2x
jMSN	34%	80%	2.4x
GanttProject	8%	21%	2.6x
FreeMind	10%	53%	5.3x

Table 4: Size of design information displayed by original and augmented GUI editor compared to total amount of information in the dynamic view.

Finally, our last metric measures the ratio of design information that is statically editable using the original editor versus the same editor augmented with our analysis (Augmented Editor). These results are displayed in Table 4.

In the second column we see the amount of statically editable information for the original COTS editor. This is the same data as shown in Table 3 because all of the information displayed by standard editors is statically editable. The third column measures the amount of statically editable information using the augmented editor. This was determined by intersection of static and dynamic models as described in Section 5 by following our editing constraints.

In the third column we see the ratio of editable design information using our approach versus the standard approach. The additional information appears because the dynamic analysis has "reattached" truncated sub-trees so they are reachable from the root main window. Our approach didn't "create" this new design information, we simply have a way of locating it and placing it in context.

The important consideration here is to judge whether the improvement we have made in these cases is of practical significance. For the experiment we have done, 3 out of 5 of the applications show at least 2x more design information editable in the context of the main window. Essentially, if a developer would like to change any of this additional information it can now be from a single conceptual module. Furthermore, the two larger applications appear to benefit the most. We believe this gives some initial evidence that our approach might help improve view-based editing even as application sizes scale and complexities are introduced.

7. Discussion, Future Work

A common question to ask about this approach is, "What if the developer wants to edit only a single custom widget instance without affecting other widget instances of the same class?". We have not supported this in our current implementation and it would not be compatible with current GUI editor methodology. Although as explained in Section 3.3.2, edits to framework widgets can be applied to the code of a custom class which created them, so that framework widgets are distinguished. Here we summarize four common situations when view-editing a custom widget in a dynamic context is and is not useful, to make sense of this question.

First, in the case that the custom widget to be edited is a Singleton [8], our approach could allow the developer to edit the design. However, using existing editors, this singleton object might not be displayed.

Second, in the case that the developer wants to edit design information and also the developer would like all widget instances of the class to be changed in the same way, then this is supported by our approach (such as in the example of changing chess piece sizes uniformly). In the traditional approach, there might be no widgets of that class displayed at all.

Third, in the case the developer wants to edit dynamic information (such as in the column label example), this would not be supported by either approach. Also there are fundamental reasons why this could not always be possible [5, 21].

The final case is one where the developer would like to change some widget's design information without affecting some other widget's from the same class, and this is not a singleton. In this case, all widgets of the class currently share this design information, because otherwise it would not have been resolved statically. However, now the developer has decided to partition the set of instances created by the class into two sets: one with the previous design and one with the new design. This requires some way to disambiguate which instances will belong to which set. This can't always be provided from the editor view, since editors do not support any kind of conditionals. In some (but not all) cases, these sets of widgets could be distinguished based on what "new" statement was used to create them, rather than which class. We leave this "context-sensitive" editing of custom classes to future work.

Although we have not quantified how often each of these situations arise, we feel that adding support for only the first two situations is a significant contribution.

8. Related Work

8.1 Software Engineering for GUI

Creating good graphical user interfaces requires considering the appearance, careful design for usability, and providing for the concrete implementation. Our proposed contribution is to make an improvement at the implementation level so we limit our discussion of related work to projects with similar goals.

Atif and IsHan et al. discuss "GUI Ripping" [22], a dynamic process in which the software's GUI is automatically "traversed" by opening all its windows and reverse engineering all widgets in a GUI forest. This information can be used as feedback for automatically generated GUI test cases [23]. Compared to our research, this other research is not concerned with view-based maintenance of GUIs. In general, maintenance tasks could require both human guided code inspection and automated test cases. View-based navigation addresses the first whereas their research addresses the second. Their research also does not address tool support for improving WYSIWYG view-based editing.

Several papers address reverse-engineering of systems in order to evolve legacy systems to more modern user interfaces. In [24], Merlo, Girard et al. discusses a method for reverse engineering user interfaces to turn console-based text interfaces into GUIs. Some researchers [2, 3] have a similar approach for migrating systems to the web. In contrast, our research has investigated understanding and making changes to an interface and not porting an interface to another platform.

Staiger et al. [29] use static whole-program analysis for reverse-engineering GUIs but these results have not been used as input to a WYSIWYG-based tool and they do not describe how the statically recovered information would be mapped back to source code changes. Dynamic and static analysis both provide different tradeoffs. Whereas dynamic analysis is precise for a particular program execution, static analysis can explore many of the potential program paths in a program. We have researched an approach where one particular dynamic view can be selected under the developer's control.

Another use of reverse-engineering for GUIs is for program comprehension and documentation. In [25], Michail introduced a tool to provide GUI-guided browsing of source. Their objective was to allow developers to find where in the code a feature was implemented, based on how code was related to the GUI. For example, to find "spell checking" code, they could locate the code which executed when the spell checking menu was selected. Compared to our research, this research does not deal with use of a WYSIWYG methodology and does not consider editing of code.

8.2 Aspects and Separation of Concerns

We make use of AspectJ to monitor a particular crosscutting concern, the GUI, at runtime. Several papers use the example of "display updating" as a crosscutting concern for which AOP can be of use [19]. Our approach is different in that we don't propose to remodularize the existing code but rather provide a crosscutting view of the code for navigation and constrained editing.

Source views can be provided by code query languages [10, 12, 13]. These languages are static in nature and thus would not be able to provide view-based navigation from dynamic views. Their use applies to a wider range of concerns than our approach and we believe our approach to be more domain-specific.

In their research on "mixin"-style languages for component-based software development, Eide et al. [6], point out that although OO design patterns are implemented through dynamic collaborations of objects, these collaborations are often statically determined. In these cases, the flexibility of dynamism can get in the way of program comprehension and potential optimizations. Similarly, we have noticed that although large parts of GUI designs are static, they can be hard to recover using existing tools.

Task-centric IDEs [18] and Concern Graphs [33] provide a projection of software development artifacts which are relevant for a particular programmer task or program feature. We see the dynamic views of a GUI as natural conceptual modules for GUI maintenance tasks. Since the conceptual modules don't always align with source modules, tool support such as we proposed could be helpful.

8.3 Software Maintenance

One difficulty in software development is that the relationship between observed program behavior and source code is not always clear. To recover the relationships between program statements for aiding maintenance tasks, both static [11, 36] and dynamic slicing [1] have been used. These approaches provide a link between a source statement (the criterion), and other statements it may depend on (static), or did depend on for some particular execution (dynamic). In contrast to our approach, we provide a link between the GUI view and the source. We do not provide links between source statements, so either static or dynamic slicing could be useful for GUI maintenance tasks in conjunction with our proposed tool support.

Other work in the area of round-trip engineering for object-oriented models (e.g. UML) [27] also must tackle with the issues of synchronizing source code with other software artifacts. Although there are similar problems, the problem domain of GUI maintenance has required research into distinct solutions.

9. Conclusion

We have discussed view-based maintenance for graphical user interfaces. One difficulty in maintaining GUIs is that the relationship between the view and source code is not always clear. We showed that one reason is because user interface code is spread across the decomposition of applications. In a case study we saw only 1 out of 5 dynamic views was built out of less than 10 classes. A popular approach to develop and maintain GUIs is to use "What you see is what you get" editors. They allow developers to work directly with user interface views instead of the scattered source elements. However, in our study a typical design view was less than 50% of the dynamic view in 4 out of 5 cases. In this paper we investigated the combination of a hybrid dynamic and static approach to allow for view-based maintenance of GUIs. Dynamic analysis provides a concrete context in which maintenance can be performed, while static checking ensures that changes propagated from the view to the source are predictable. Our approach enabled at least 50% of the dynamic view to be editable in 4 out of 5 cases. We showed an addition of between 1.2x and 5.3x more design information in 4 of the 5 cases we looked at. Furthermore, the two larger applications appeared to benefit the most.

Acknowledgements

This work was supported by a grant from IBM Centers for Advanced Study.

References

[1] Agrawal, H. and Horgan, J. R. Dynamic program slicing. In *Proc. of the Conference on Programming Language Design and Implementation*, 1990.

[2] Bodhuin, T., Guardabascio, E. and Tortorella, M. Migrating COBOL Systems to the WEB by Using the MVC Design Pattern. In *Proc. of the Working Conference on Reverse Engineering*, 2002.

[3] Boldyreff, Cornelia and Kewish, Richard. Reverse Engineering to Achieve Maintainable WWW Sites. In *Proc. of the Working Conference on Reverse Engineering*, 2001.

[4] CrosswordSage V0.3.5, http://crosswordsage.sourceforge.net/, Visited 10/6/2007.

[5] Dayal, U. and Bernstein, P.A. On the correct translation of update operations on relational views. 1982, Transactions on Database Systems, Vol. 7, pp. 381-416.

[6] Eric Eide and Alastair Reid and John Regehr and Jay Lepreau, Static and Dynamic Structure in Design Patterns, In *Proc. of the International Conference on Software Engineering*, 2002.

[7] FreeMind V0.8.0,http://freemind.sourceforge.net/wiki/index.php/Main_Page, Visited 10/6/2007.

[8] Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, hardcover, 395 pages, Addison-Wesley.

[9] GanttProject V2.0.1, http://sourceforge.net/projects/ganttproject/, Visited 10/6/2007.

[10] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with DataLog. In *Procs. of the European Conference on Object-Oriented Programming*, 2006.

[11] Susan Horwitz and Thomas W. Reps, The Use of Program Dependence Graphs in Software Engineering. In *Proc. of the International Conference on Software Engineering*, 1992.

[12] Janzen, D. and De Volder, K. Navigating and Querying Code Without Getting Lost, In *Proc. of the International Aspect-Oriented Software Development Conference*, 2003.

[13] Janzen, D. and De Volder, K. Programming with Crosscutting Effective Views. In *Proc. of the European Conference on Object-Oriented Programming*, 2004.

[14] Java Chess (JChess) V1.01a3, Harald Faber, Andreas Rueckert, Thomas Dalichow, http://www.java-chess.de/, Visited 10/6/2007.

[15] Jigloo SWT/Swing GUI Editor for Eclipse and WebSphere, CloudGarden V4.0.2, http://www.cloudgarden.com/jigloo/, Visited 10/6/2007.

[16] JMSN V0.9.9b2, http://sourceforge.net/projects/jmsn/, Visited 10/6/2007.

[17] Alan C. Kay. The early history of Smalltalk, In HOPL-II: *The second ACM SIGPLAN conference on History of programming languages*, 1993.

[18] Mik Kersten and Gail Murphy. Mylar: a degree-of-interest model for IDEs. In *Proc. of the International Conference on Aspect-oriented Software Development*, 2005.

[19] Kiczales, G and Mezini, M. Aspect-oriented programming and modular reasoning. In *Proc. of the International Conference on Software Engineering*, 2005.

[20] Lorenz, D., Vlissides, J. Designing Components versus Objects: A Transformational Approach In *Proceedings of the International Conference on Software Engineering*, 2001.

[21] McCarthy, J. Automata Studies. Princeton University Press, 1956, pp. 177-181.

[22] Memon, A., Banerjee, I. and Nagarajan, A. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proc. of the Working Conference on Reverse Engineering*, 2003.

[23] Memon, A. and Soffa, M. Regression testing of GUIs. In *Proc. of the Symposium on Foundations of Software Engineering*, 2003.

[24] Merlo, Ettor. Reengineering User Interfaces. IEEE Software, Vol. 12, pp. 64–73.

[25] Michail, A. Browsing and searching source code of applications written using a GUI framework. In *Proc. of the International Conference on Software Engineering*, 2002.

[26] Myers, Brad A. User interface software tools. ACM Transactions on Computer Human Interaction, 1995. pp. 64–103.

[27] E. V. Paesschen, W. D. Meuter, and M. D'Hondt. Selfsync: a dynamic roundtrip engineering environment. In *Proc. International Conference on Model-Driven Engineering Languages and Systems* , 2005.

[28] S. M. Selkow. The tree-to-tree editing problem. *Inform. Process.Lett.*, 6(6):184–186, 1977

[29] Staiger, S. Static Analysis of Programs with Graphical User Interface. In *Proc. of the European Conference on Software Maintenance and Reuse*, 2007. pp. 252–264.

[30] M. Stuart. Java GUI Builders. http://www.fullspan.com/articles/java-gui-builders.html, Visited 10/6/2007.

[31] Swing Designer V6.4.1, Instantiations, http://www.instantiations.com/swing-designer/home_content.html, Visited 10/6/2007.

[32] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison Wesley, 1998.

[33] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proc. of the International Conference on Software Engineering*, 2002.

[34] Tarr, P., Ossher, H., Harrison, W. and Sutton, Stanley Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns, In *Proc. of the International Conference on Software Engineering*, 1999.

[35] Visual Editor V1.2, IBM, http://www.eclipse.org/vep/WebContent/main.php, Visited 10/6/2007.

[36] M. Weiser. Program slicing. IEEE Transactions on Software Engingeering, 10(4):352--357, July 1984.

[37] Yuan, Xun and Memon, Atif M. Using GUI Run-Time State as Feedback to Generate Test Cases. In *Proc. of the International Conference on Software Engineering*, 2007.