Automated Data Collection for Usability Evaluation in Early Stages of Application Development

YONGLEI TAO
School of Computing and Information Systems
Grand Valley State University
Allendale, MI 49401
USA

Abstract – Modern interactive applications are event-driven. User interface events are referred to as those that are generated when the user interacts with the application via its user interface. Relevant user interface events provide crucial data for usability evaluation. Due to iterative design, evaluation, and redesign of the user interface in early stages of application development, automated support for data collection and analysis is desirable. We in this paper describe an aspect-oriented approach to automated data collection and as shown in a case study the data collected with our approach is amenable to subsequent analysis.

Key-Words: - Usability Evaluation, Tool Support, Aspect-Oriented Programming, Software Engineering

1. Introduction

Usability is a key quality attribute for the success of interactive applications. It is increasingly sought to improve the productivity and to reduce the overhead costs for end-users. A practical solution to meeting the usability requirement is early and ongoing usability evaluation [1, 5].

In usability evaluation, users use an application to complete a pre-determined set of tasks. Information on user behavior with respect to the user interface is collected and analyzed to determine how well the user interface supports users' task completion. Since usability data is extremely voluminous and rich in detail, automated techniques to collect and analyze the data are required [3].

User interface development is exploratory and evolutionary in nature. Due to iterative design, evaluation, and redesign of the user interface in early stages of application development, new windows may be introduced; existing windows may be combined, split, or removed; and screen elements may be added, removed, or replaced. Changes in the user interface inevitably affect other components in the application [14]. In Java, for examples, even re-layout of screen elements requires to alter a few lines of code. Consequently, automated support for early usability evaluation must be adaptable to frequent changes in the target application.

We in this paper describe an aspect-oriented approach to automated data collection. Aspects allow collecting usability data in an adaptive and self-contained fashion without having to instrumenting the target application or its platform. We also use a case study on this approach to show that the collected data is amenable to subsequent analysis. As demonstrated in this paper, the aspect-oriented approach is effective for early usability evaluation.

The rest of this paper is organized as follows. Section 2 focuses on related work. Section 3 discusses user interface events in the context of software structure for interactive applications. Section 4 describes how to use aspects to capture various user interface events and section 5 presents a case study of our approach. Section 6 concludes this paper.

2. Related Work

Various techniques for automated data collection can be found in the literature [2]. Some of them capture user interface events at the keystroke or system level. Recording data at a lower level produces voluminous log files, making it difficult to map recorded usage into high-level tasks [3]. An alternative solution is to instrument the target program or its platform. Instrumentation tends to be distributed throughout the target code and therefore is inflexible when changes in the user interface take place frequently [6].

AOP is known as an effective technique for modularizing crosscutting concerns such as monitoring, tracing, and logging [4]. Even though it has been around for years, using AOP in support for usability evaluation is still relatively new. Tarby et. al. provide a survey on using AOP and other techniques for early usability evaluation [7], Tarta et
address areas in which AOP can be used to support automatic usability evaluation [8], and we discuss an initial investigation on the suitability of using AOP in tool support for early usability evaluation [9].

3. User Interface Events
Modern interactive applications are event-driven. User actions, such as clicking on a command button and entering text at the keyboard, cause certain operations in an interactive application to execute. As a consequence, the application updates its views to display the result or brings up dialogs for additional information from the user. User interface events are referred to as those that are either triggered by user actions or generated by the application in response to user actions [3]. Sequences of user interface events correspond to steps taken by the user in completing the intended tasks.

Interactive applications are mostly structured according to the Model-View-Controller architecture (MVC) [11]. Originally designed to support multiple views for the same data, MVC has become the central feature of interactive applications. MVC describes an application in terms of three fundamental abstractions: models, views, and controllers. Roughly, the model manages application data, the view is responsible for visual presentation, and the controller handles input events for views. By encapsulating the three abstractions into separate components, the impact of changes in the user interface is minimized and the reusability of domain objects is enhanced.

Moreover, as the user interacts with the application via its user interface, the model, view, and controller must notify each other about what has happened in order to collaborate for achieving the overall objective. For example, the controller needs to notify the model of user actions and the model has to notify the view of its state change. Separating the three components also exposes user interface events within the application, making it possible to identify and capture them.

User interface events provide crucial information for usability evaluation, enabling the developer to find indicators for potential usability problems, such as areas in which mistakes were made, unnecessary or undesirable steps were taken, and extra assistances (such as the undo operation and on-line help) were requested. For example, the failure to following a navigational path as expected may indicate the lack of visual clues for what the user needs to know. Additional data, such as error rates, task frequencies, and range of functions used, is also useful.

4. Capture User Interface Events
A concern is a specific requirement or consideration that must be addressed in order to satisfy the overall objective. While core concerns reflect the basic functionality of a module in an application, crosscutting concerns span multiple modules [4]. Aspect-Oriented Programming (AOP) provides an effective way to modularize crosscutting concerns. AspectJ is an aspect-oriented extension to the Java programming language [4]. It provides a set of syntactic constructs to specify crosscutting concerns. In AspectJ, an aspect is composed of pointcuts and advices. Join points are well-defined points within an application, such as a method call made in program execution. Pointcuts are a means of referring to collections of join points and certain values at those join points. A advice is a piece of code that defines actions, for example, printing information of a value that the advised joint point refers to. With an aspect present, additional behavior is introduced at specified points within the target application.

User interface events are generated as natural products of the normal operation of an interactive application. Event-capturing is clearly a crosscutting concern. We use AspectJ to capture relevant user interface events.

4.1 Common Event-Capturing Behavior

```java
public abstract aspect AbstractTraceAspect {
    static DataSpoolerClass dataSpooler =
        new DataSpoolerClass();

    public void report(String message)
    {
        String msg = message + " at " +
        System.currentTimeMillis() / 1000.0;
        dataSpooler.writeLine(msg);
    }
}
```

Fig. 1 An Abstract Aspect

Shown in Fig. 1 is the definition of an abstract aspect that serves as the base for all event-capturing aspects. Aspect AbstractTraceAspect defines a method for reporting an event, including its
description (given as the parameter) and occurrence
time in seconds. Here, a static instance of
DataSpoolerClass acts as an accumulator to record
captured data. Due to limited space, we omit the
definition of DataSpoolerClass.

4.2 View-Updating Events
In the MVC architecture, when the model changes
state, all of its views are notified and updated to
reflect the change. A well-known design pattern,
Observer, describes an effective way to establish
such a one-to-many dependency [12]. Java provides
class Observable and interface Observer to
implement the Observer pattern. Briefly, the model
is an Observable and a view is an Observer. Each
view defines its own update() method to keep its
display up to date and the model notifies its views
by invoking their update() methods. As such, a call
to the update() method for an object of any class in
the Observer-based class hierarchy indicates the
occurrence of a view-updating event. We define an
aspect ViewUpdateTraceAspect as shown in Fig. 2
to capture view-updating events.

```java
public aspect ViewUpdateTraceAspect extends AbstractTraceAspect {
    pointcut viewUpdatePointcut (Object obj): execution (void
    Observer+.update (Observable, Object))
    && args (Observable, obj);
    before(Object obj): viewUpdatePointcut (obj)
    { report (thisJoinPoint.getThis() + " for " + obj + " Updated"); }
}
```

Fig. 2 Apect for View Updating Events

A aspect ViewUpdateTraceAspect extends AbstractTraceAspect and defines a pointcut
viewUpdatePointcut() to capture joint points that make a call to the update() method for a view. It
defines a piece of advice to identify the model
that fires the event. Here, viewUpdatePointcut() takes an event argument from its constituent joint
point and passes it to the advice, allowing the advice
to find information about the model.

In the declaration of viewUpdatePointcut(),
Observer+ means any class that implements the
Observer interface, including both the current and
potential ones. Hence, the introduction of a new
view or removal of an existing view has little
impact to aspect ViewUpdateTraceAspect.

4.3 Handler-Notifying Events
In an interactive application, a user’s action, such as
clicking a button or entering data at the keyboard,
fires an event and an event handler (i.e., listener in
Java) is notified of that event. In an event-driven
system, an event is an instance of a predefined event
type and an event handler is defined according to a
standard protocol. A such a consistent treatment for
events of a specific type is achievable.

In a Java program, when the user clicks a button
or a menu item, an action event is fired. A handler
that implements the ActionListener interface
is notified by a call to its actionPerformed() method.
In other words, a call to method actionPerformed()
indicates the occurrence of an action event.

Similarly, when the user enters data into a text
field, a key event is fired. A handler that
implements the KeyListener interface is notified by
a call to its KeyTyped() method. Hence, a call to
KeyTyped() indicates the occurrence of a key event.

```java
public aspect ActionEventTraceAspect extends AbstractTraceAspect {
    pointcut actionEventPointCut (ActionEvent e)
    : execution (void
    ActionListener+.actionPerformed (ActionEvent))
    && args(e);
    before(ActionEvent e) : actionEventPointCut (e)
    { report(e.getSource().getClass().getName() + " " + e.getActionCommand() + " Clicked "); }
}
```

```java
public aspect KeyEventTraceAspect extends AbstractTraceAspect {
    pointcut captureKeyEvent (KeyEvent e)
    : execution (void
    KeyListener+.keyTyped(KeyEvent))
    && args (e);
    before(KeyEvent e): captureKeyEvent (e)
    { report("" + e.getKeyChar() + " Typed in " + e.getSource().getClass().getName()); }
}
```

Fig. 3 Aspects for Action and Key Events

We define aspects to capture action and key
events respectively. As shown in Fig.3, aspect
ActionEventTraceAspect declares a pointcut to
capture joint points that make a call to method actionPerformed() for any handler that implements the ActionListener interface. Similarly, aspect KeyEventTraceAspect captures calls to method KeyTyped() for any handler that implements the KeyListener interface. Both of them also contains a piece of advice that receives an event object from the advised joint point and uses it to identify the event source.

We can use such an aspect to capture action events without having to worry about from which screen elements they originate or by which handlers they are handled. Changes in the interface with respect to the event source, such as adding or removing a button, will certainly affect some handler classes. However, there is little impact to the aspect responsible for action events. Similar arguments can also be made for the aspect responsible for key events.

In addition to action and key events, other events also occur when the user interacts with the application’s user interface, such as mouse events. Java provides a listener interface for each type of input event. Similarly, handler classes and aspects can be specifically defined to capture them. Adding a new aspect does not affect the existing ones. While action and key events provide usability data at the task level, other events such as mouse events contribute primarily to usability information at a lower level of abstraction [9]. Use of select aspects allows us to address usability issues that early usability evaluation focuses on.

4.4 Other User Interface Events
Interactive applications often use standard dialogs to communicate with users, such as displaying information, error, and warning messages. In Java, class JOptionPane provides methods for bringing up different dialogs as needed, such as showConfirmationDialog(), showMessageDialog(), and showOptionDialog(). Apparently, a call to a “show” method of JOptionPane indicates a dialog-displaying event. We define an aspect to capture such an event.

In addition, exceptions are also of interest to the developer because it indicates an abnormal situation often caused by user actions. We define an aspect by using a predefined pointcut, called handler(), to capture exception-throwing events.

Note that either type of events can be generated from any part of an interaction application. With aspects, we can capture them in a uniform way without having to have specific knowledge about all the participating objects. Due to limited space, we omit the declarations of these two aspects. Interesting readers can contact the author for a complete listing.

5. A Case Study
Here we use a GUI (Graphical User Interface) application AccountManager, as a case study on our approach to automated data collection. We adopted the code from [10] and enhanced it with a more complete list of user interface events.

5.1 Application Description
A application AccountManager is a Java program structured according to the MVC architecture. Briefly, this application allows the user (e.g., a bank customer) to manage several accounts that one owns. As shown in Fig. 4, a text view and a bar graph view are provided for each account to display the account information. A pie chart view is provided to display the customer’s total assets held in both accounts. Also a text field and two buttons are provided for each account, where the former allows the user to enter an amount and the latter to withdraw and deposit the input amount, respectively.

![Fig. 4 Example User Interface](image)

In application AccountManager, class Account extends the Observable class and three view classes (namely PieChartView, TextView, and BarGraphView) implement the Observer interface. An Account notifies its three views by invoking the update() method. Method update() of each view is
responsible for bringing itself up to data. As discussed in the above section, a view-notifying event occurs whenever a call to the update() method for a view is made.

Also included in this application are two handler classes. Class ButtonHandler is responsible for action events which occur when the user clicks a button in order to execute a transaction. Class KeyHandler is responsible for key events which occur when the user enters an amount into a text field. A ButtonHandler is notified of an action event when its actionPerform() method is invoked while a KeyHandler is notified of a key event when its keyTyped() method is invoked. As discussed in the above section, a handler-notifying event occurs whenever a call to either the actionPerform() for a ButtonHandler or keyTyped() for a KeyHandler.

Application AccountManager provides a JOptionPane for the user to confirm any change to one’s account and uses exception handling in case of errors.

### 5.2 Recorded Data

When application AccountManager runs with the aspect code, user interface events are generated by the application and recorded by the aspect code. As illustrated in Fig. 5, recorded data shows what happened during program execution.

### 5.3 Preprocessed Data

We create a preprocessing program for extracting useful information from the recorded data. Because of being produced by the above-mentioned aspects, recorded data follows several basic patterns. Our preprocessing makes use of Java’s pattern matching capability to extract useful information.

![Fig. 6 Preprocessing Data](image)

Shown in Fig. 6 is what is extracted by our preprocessor from the recorded data given in Fig. 5. Preprocessed data is readily usable and also amendable for further analysis either by the evaluator directly or by an automatic tool.

Usability evaluation is to find out to which degree the interface support the effective and efficient completion of selected tasks. As such, tasks are the basic unit of data for usability evaluation.

We can identify instances of tasks the user performed from the preprocessed data in Fig. 6.
Shown below, for example, is a successful instance of the Deposit task:

- **Key 1 Typed in JTextField at 6087.6**
- **Key 0 Typed in JTextField at 6087.9**
- **Key 0 Typed in JTextField at 6088.2**
- **JButton Deposit Clicked at 6090.9**
- **Dialog Shown: Confirmation at 6090.9**
- **AssetPieChartView for Account 1 Updated at 6092.3**
- **AccountBarGraphView for Account 1 Updated at 6092.3**
- **AccountTextView for Account 1 Updated at 6092.3**

And given below is an erroneous instance of the Deposit task:

- **Key 0 Typed in JTextField at 6109.9**
- **JButton Deposit Clicked at 6112.8**
- **catch(NumberFormatException)in class AccountController at 6112.9**
- **Dialog Shown: Invalid Amount at 6112.9**

With automated support, data at the task level can be identified according to well-defined tasks in the requirements specification. A collection of task-level data provides the basis for subsequent activities in usability evaluation.

### 6. Summary
Adaptability is crucial for automated support for early usability evaluation. We use the MVC architecture to expose user interface events from within an interactive application. We also use Java's interface construct to specify a standard way of notifying an object of an event occurrence, against which an aspect to capture that event is defined. Using interfaces to specify crosscutting concerns makes possible to address them without knowing or caring what class that object belongs to [13]. As a consequence, the aspect code becomes independent of specific features of the user interface and therefore is adaptable to possible changes.

It is worth noting that although the above case study is written in Java, our approach (based on the notions of MVC, interface/abstract classes, and aspects) are language independent and therefore can apply to a variety of interactive applications.

Our future research focuses on automated support for extracting task-level information and analyzing the information in order to build a more usable product.

### References: