Toward Computer-Aided Usability Evaluation for Evolving Interactive Software

(Position Paper for RAM-SE '07)

Yonglei Tao
School of Computing and Information Systems
Grand Valley State University
Allendale, Michigan, USA

Abstract – Recurrent redesign on an application's user interface is driven by changing requirements, user profiles and experiences, as well as technologies. User interface evolution also has impact to the application itself, which imposes a great challenge on providing tool support to ensure a smooth transformation in this process. We in this paper explore the suitability of using an aspect-oriented approach to computer-aided usability evaluation. Using aspects, a support tool is not only flexible for collecting data to address diverse usability considerations in the evolution process but also adaptable to continuous changes in the application. We also discuss our future research on other relevant issues about such a tool.

1. Introduction

Interactive software evolves along one or more dimensions during its lifetime, including functionality, architecture, code, and user interface. Changes in one dimension often affect, interact, and impact others [1]. As such, the evolution of an interactive application imposes a great challenge not only on developing the application itself but also on providing tool support to ensure a smooth transformation in this process [2].

Usability is a key quality attribute for the success of interactive applications. A practical solution to building a usable product is early and ongoing usability evaluation [3]. 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 captured and analyzed to determine how well the user interface supports users' task completion. Since evaluation activities such as data collection and analysis are very time-consuming, tool support is indispensable [5, 6].

Recurrent redesign on an application's user interface is driven by changing requirements, user profiles and experiences, as well as technologies. Usability considerations vary in the process of user interface evolution [4]. As a new feature is introduced, for example, the attention is focused on the flow of the basic user-system interaction, including the coordination of data exchange between the user and the system as well as the navigational structure. While a new interaction style is made available to support a particular user group for effective use of the application, however, usability considerations largely reflect on physical, spatial, and visual characteristics of screen elements. A support tool must be flexible for collecting data at a level of abstraction that is appropriate to address specific needs in different stages of the evolution process.

As the user interface evolves, 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 various components in the application [7]. In Java, for examples, even re-layout of screen elements requires to alter a few lines of code. Hence, a support tool must also be adaptable to continuous changes in the application.

In this paper, we explore the suitability of using an aspect-oriented approach to computer-aided usability evaluation. We describe how to use aspects to capture user interface events that occur when the user interacts with an application's user interface. Using aspects for data collection paves the way for analyzing the acquired data and identifying potential usability problems. We also discuss our future research on other relevant issues about such a tool.

2. Related Work

AOP (Aspect-Oriented Programming) is known as an effective way of modularizing crosscutting concerns such as monitoring, tracing, and logging [8, 10]. As far as we know, using an aspect-oriented approach to computer-aided usability evaluation, however, does not seem to have received as much attention as it should have been, in part due to the gap between the communities of SE (Software Engineering) and HCI (Human-Computer Interaction).

Proposals about automatic techniques for capturing user interface events can be found in the literature [5]. Some of those techniques capture events at the keystroke or system level regardless of the usability issues under consideration. Recording data at that level produces voluminous log files and makes it difficult to map recorded usage into high-level tasks [6]. Usability-related information can also be obtained by instrumenting the target program or its platform. Because such information does not appear at one particular place, instrumentation in a traditional way tends to be distributed throughout the target code [9]. Obviously, techniques as such are inappropriate when changes in an application occur quite often. Adaptive techniques, such as AOP, are more promising for our purposes [10, 11].

Java-style interfaces enhance, facilitate, and even make possible the flexibility, modifiability, and extensibility that are highly desirable in object-oriented design [12]. Interfaces can also improve the quality of aspect-oriented design [13]. We use interfaces to expose crosscutting behavior against which aspects are defined. Using interfaces and aspects jointly provides the benefit of adaptability for automatic support for usability evaluation.

3. The MVC Architecture for Interactive Applications

The Model-View-Controller architecture (MVC) was originally designed for applications that provide multiple views for the same data [15]. It has gradually become the central feature of modern interactive applications. Based on the object-oriented principles, 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, MVC minimizes the impact of user interface changes and increases the reusability of domain objects.

User interface events are generated as natural products of the normal operation of an interactive application, including input events (such as the user clicking on a command button) and output events (such as the application bringing up a message box). Sequences of events result from steps taken by the user in completing tasks. In MVC, the view and controller take appropriate actions when they are notified of corresponding events. Separating the three abstractions also exposes user interface events within the application.

We in this paper use a GUI (Graphical User Interface) application AccountManager, adopted from [14] with modifications, as an example. Briefly, the application is a Java program intended to manage several bank accounts for a customer. 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 all the 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.

3.1 Model and View

Figure 1 is a UML class diagram that illustrates key classes in the model and view of the application's MVC architecture. As shown in Figure 1, class Account is a model and classes PieChartView, TextView, and BarGraphView represent three views for an Account.

When an account 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 [16]. We use Java's Observable class and Observer interface to implement the Observer pattern. As shown in Figure 1, class Account extends class Observable and the three view classes (i.e., PieChartView, TextView, and BarGraphView) implement interface Observer.

Note that each view defines its own update() method to refresh its display and the account notifies its views by invoking their update() methods. As a result, a call to the update() method for an object of any class in the Observer-based class hierarchy indicates the occurrence of an output event.

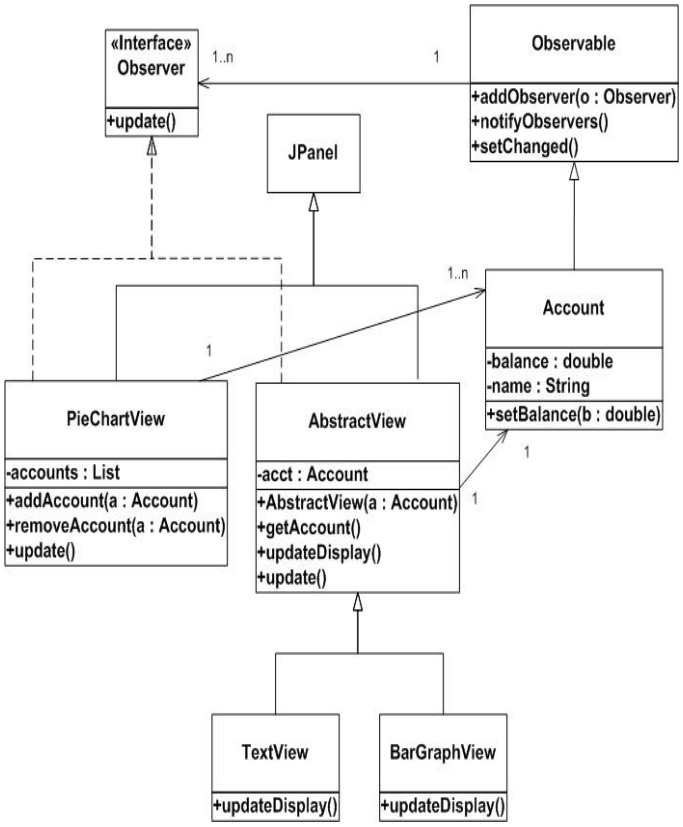

Figure 1: Model and View

3.2 Model and Controller

Due to limited space, we omit the UML class diagram for the model and controller. In Java, a listener class is responsible for handling an input event. Application AccountManager defines two listener classes, one for executing a transaction for an account when a button is clicked and the other for validating the user input entered in a text field. Both listener classes handle the same type of input events, that is, action events in Java. Action events originate from the user's actions with respect to screen elements such as buttons, menu items, and text fields. Java provides an ActionListener interface with method actionPerformed() for handling action events. Hence, the two listener classes must implement the ActionListener interface.

Note that each listener class has to define its own actionPerformed() method for handling an action event. Application AccountManager notifies a listener of the user's action by invoking its actionPerformed() method. As a result, a call to the actionPerformed() method for an object of any class that implements the ActionListener interface indicates the occurrence of an input event in this application.

4. Data Collection with Aspects

AspectJ is an extension to the Java programming language [10]. It provides constructs to modularize crosscutting concerns that would otherwise result in code scattered over multiple modules. We use the aspect construct to capture user interface events.

Java-style interfaces are essential to adaptability. An interface is a collection of method signatures. It defines a standard protocol to interact with an object without knowing or caring about what class that object belongs to. In application AccountManager, interface Observer specifies a standard way for a model to notify its views and interface ActionListener for the application to notify an event listener. We expose crosscutting concerns of interest through interfaces against which aspects are defined. Using interfaces allows us to specify crosscutting behavior without being committed to a particular class hierarchy. As a result, the dependency of the aspects code on specific features of the user interface is loosened.

4.1 Capturing Output Events

In application AccountManager, each view is updated when being notified of state change in its model. Such a notification is made through a call to the update() method for each view. Aspect UpdateView, as declared below, is intended to capture output events that occur when views are notified.

```
import java.awt.*;
import java.util.*;

public aspect UpdateView {
    // Pointcut Declaration
    pointcut traceUpdate ( Object obj )
            : cflow ( execution (
                    void Observer+.update (
                        Observable, Object )))
            && args (Observable, obj);

    // Advice Definition
    after ( Object obj ): traceUpdate ( obj ) {
            System.out.println ( obj +
                            " view updated " );
    }
}
```

Aspect UpdateView defines a pointcut traceUpdate() to capture joint points that make a call to the update() method for a view. It also defines a piece of advice to identify the account whose state change causes the output event. Here, pointcupt traceUpdate() takes an event argument from the advised joint point and passes it to the advice, which gives the advice the information it needs.

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

4.2 Capturing Input Events

When the user clicks a button or enters data in a text field, a listener is notified of the action event. We define an aspect to capture action events. Basically, this aspect contains a pointcut to capture joint points that make a call to method actionPerformed() for a listener. It 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 user interface with respect to the originating screen elements of the action event, such as adding or removing a button, will affect the related handler classes. But they won't have much impact to that aspect and it will continue to function as it is specified.

Note that other types of input events also occur when the user interacts with the application's user interface, such as mouse and key events. Java provides a listener interface for each type of input event. Similarly, handler classes and aspects can be specifically defined to capture other types of input events. Such an addition does not affect the existing ones in any way. While action events contribute to usability information at the application level, mouse and key events contribute primarily to usability information at a lower level of abstraction. Use of those aspects selectively would allow us to address different usability considerations.

When application AccountManager runs with the aspects described above, a list of input and output events will display on the screen, showing which button is clicked, which view is updated, and so forth. Such a list of events provides the basis for the follow-up activities in usability evaluation.

5. Summary and Future Research

As demonstrated by the above example, the aspect-oriented approach is suitable for building a support tool for usability evaluation. Using aspects not only provides the advantage of flexibility but also offers the benefit of adaptability. In addition, using aspects makes possible not only to collect usability-related information from the captured events but also to obtain relevant information available elsewhere in the application, which is helpful for a meaningful interpretation of certain data. Compared with some of the existing techniques that require an additional step to extract appropriate information from the raw data, the aspect-oriented approach is more effective.

It is worth noting that although we use Java in the example application, our approach, which is based on the notions of MVC, interfaces/abstract classes, and aspects, is language independent.

In addition to data collection, it is equally important to provide tool support for data analysis. Analyzing the acquired data manually would be difficult and tedious without tool support. In addition to data collection, relevant issues as listed below require further research:

(1) Identifying tasks and sequences of tasks that the user is intended to accomplish from the acquired data. User interface design is centered on tasks (or use cases) [3]. As a consequence, tasks are a natural unit of data for analysis purposes. Well-defined tasks in the requirements specification provide a basis for identifying tasks. Here, it is important to separate application-specific knowledge from general processing logic for the sake of adaptability.
(2) Analyzing data obtained from multiple users to measure usability attributes of a user interface and to identify potential issues affecting them. Examples of quantitative measures include time to complete a task, task frequencies, range of functions used, and number of errors or repeated errors. More importantly, analyzing the acquired data enables us to find indicators for potential usability problems, for example, areas in which mistakes were made, unnecessary or undesirable steps were taken, and extra assistances (such as undo and on-line help) were requested. Failure for a (group of) user to follow a navigational path as expected may indicate

the lack of adequate visual clues for what the user needs to know. Often, whether or not visual guidance is adequate depends on the user who uses the application. Here, a challenge is to find out to which user group it is adequate and to which one it is not. We will investigate use of data mining techniques in this regard.

In addition, AspectJ is classified as a static AOP system. Static AOP systems allow weaving in aspects at compile or load-time. On the other hand, the dynamic AOP systems allow weaving aspects in at run-time. As a result, programmers can dynamically plug and unplug an aspect in/from running software [17]. Obviously, a dynamic AOP system seems to be more appropriate for our purposes. It is also an interesting issue that deserves future attention.

Reference:

[1] M. Lehman and J. Ramil, “Evolution in Software and Related Areas”, the 2001 International Workshop on Principles of Software Evolution (IWPSE), Vienna, Austria.

[2] Harald Gall and Michele Lanza, “Software Evolution: Analysis and Visualization”, the 2006 International Conference on Software Engineering (ICSE), May 20-28, 2006, Shanghai, China.

[3] R. J. Torres, “Practitioner’s Handbook for User Interface Design and Development”, Pretice-Hall, 2002.

[4] Xavier Ferre, et al., “Usability Basics for Software Developers”, IEEE Software, January/February 2001, pp.22-29.

[5] Melody Ivory and Marti Hearst, “The State of the Art in Automating Usability Evaluation of User Interfaces”, ACM Computing Survey, Vol. 33, No. 4, December 2001, pp. 470-516.

[6] David Hilbert and David Redmiles, “Extracting Usability Information from User Interface Events”, ACM Computing Surveys, 384-421, Vol. 32, No. 4, Dec. 2000.

[7] Maria Francesca Costabile, et al., "Supporting Interaction and Co-evolution of Users and Systems", the 2006 International Conference on Advanced Visual Interfaces (AVI), May 23-26, 2006, Venezia, Italy.

[8] Tina Low, “Designing, Modeling and Implementing a Toolkit for Aspect-oriented Tracing (TAST)”, Workshop on Aspect-Oriented Modeling with UML, AOSD ’02, April 2003, Univ. of Twente, The Netherlands.

[9] Morgan Deters and Ron Cytron, “Introduction to Program Instrumentation using Aspects”, Proceedings of the ACM OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay, Florida, USA, Oct. 2001.

[10] Ramnivas Laddad, “AspectJ in Action: Practical Aspect-Oriented Programming”, Manning Publications Co., 2003.

[11] Yonglei Tao, “Capturing User Interface Events with Aspects", Proceedings of the 12th International Conference on Human-Computer Interaction, LNCS 4553, pp.1171–1180, J. Jacko (Ed.), Springer-Verlag Berlin Heidelberg 2007.

[12] Peter Coad and Mark Mayfield, “Java Design: Building Better Apps & Applets”, Prentice Hall, pp.223-288, 1999.

[13] William Griswold et al., “Modular Software Design with Crosscutting Interfaces”, IEEE Software, January/February 2006, pp. 51 – 60.

[14] H. M. Deitel, et al., “Advanced Java 2 Platform: How to Program”, Prentice-Hall, pp. 85-134, 2002.

[15] Glenn Krasner and Stephen Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, JOOP, Aug. / Sept. 1988.

[16] Erich Gamma, et al., “Design Patterns: elements of Reusable Software Architecture”, Addison-Wesley, 1995.

[17] Yoshiki Sato, et. al., “A Selective, Just-In-Time Aspect Weaver”, the 2nd International Conference on Generative Programming and Component Engineering (GPCE ‘03), LNCS 2830, Springer-Verlag, 2003.