Annotating UI Architecture with Actual Use

Neil Ramsay, Stuart Marshall, Alex Potanin

School of Mathematics, Statistics, and Computer Science
Victoria University of Wellington,
PO Box 600, Wellington, New Zealand,
Email: {ramsayneil | stuart | alex}@mcs.vuw.ac.nz

Abstract

Developing an appropriate user interface architecture for supporting a system's tasks is critical to the system's overall usability. While there are principles to guide architectural design, confirming that the correct decisions are made can involve the collection and analysis of lots of test data. We are developing a testing environment that will automatically compare and contrast the actual user interaction data against the existing user interface architectural models. This can help a designer more clearly understand how the actual tasks performed relate to the proposed architecture, and enhances feedback between different design artifacts.

Keywords: Usage centered design, user interaction, automated user testing, user interface event monitoring.

1 Introduction

Several processes have been suggested for designing usable user interfaces, including – amongst others – user-centered design and usage-centered design. These approaches tend to be iterative, so designing user interfaces involves significant amounts of testing and revision to achieve a satisfactory solution.

We are interested in supporting the evaluation of user interface prototypes in Constantine and Lockwood's usage-centered design process (Constantine & Lockwood 1999). This involves using the artifacts that are first created during the early stages of usage-centered design (and that are subsequently evolved into a solution), and combining them with usage data extracted automatically from users' using the resulting prototype systems.

In particular, there are six principles that Lockwood and Constantine suggest should be the core of any good user interface: structure, simplicity, visibility, feedback, tolerance and reuse. We focus on two of these six principles: *structure* and *visibility.*

The Structure Principle is defined as "organising the user interface in meaningful and useful ways ... putting related things together and separating unrelated things".

The Visibility Principle is defined as "keeping all needed options and materials for a given task visible without distracting the user with extraneous or redundant information".

We are developing tool support to help identify — from actual use — whether a proposed architectural design satisfies these two principles.

The contributions of this paper are:

1. An overall process for helping usage-centered designers evaluate their architectures for appropriate structure and visibility.
2. A prototype model extraction tool for Java-implemented prototypes.
3. A discussion on our ongoing development of tool support to annotate design models with implementation results.

2 Design Models

Designers already create models and artifacts as a consequence of following usage-centered design. These models are used to document the users and their tasks, and help determine the properties and purpose of various tools and materials within the prototype's interface.

In usage-centered design, the architecture of the user interface is represented by abstract prototypes (often *canonical abstract prototypes*) (Constantine 2003) and *navigation maps* (Constantine & Lockwood 1999).

These models are based on the concept of *interaction spaces* (which can be considered to be such things as windows, dialog boxes, or pages in a tabbed view), and are commonly designed so that each interaction space supports either a single task or a small set of related tasks, and so that each task can be achieved using only a few interaction spaces. These guidelines are derived from the structure and visibility principles referenced earlier.

One question that could legitimately be asked of the architectural design models (especially in later iterations as changes are made) is whether the models are still adhering to these principles.

2.1 Canonical Abstract Prototypes

Canonical Abstract Prototypes are architectural models of the user interface that describe the nature and general layout of components in a particular interaction space, while hiding details such as the look and feel of the components' implementations.

The model is made up of *universal abstract components* (hereafter referred to as *abstract components*) representing a "specific abstract interaction function". Examples of this include components for creating new records or objects, components for starting activities, and components for displaying a collection of data elements. There are twenty three abstract components, split into three categories: tools, materials, and active materials (the latter essentially being those abstract components that can be considered a hybrid of tools and materials).

Each of these twenty three component types has an implementation-independent visual notation.

An example canonical abstract prototype for the Thunderbird Address Book along with a screenshot of the actual implemented interface can be seen in figures 1 and 2 respectively.

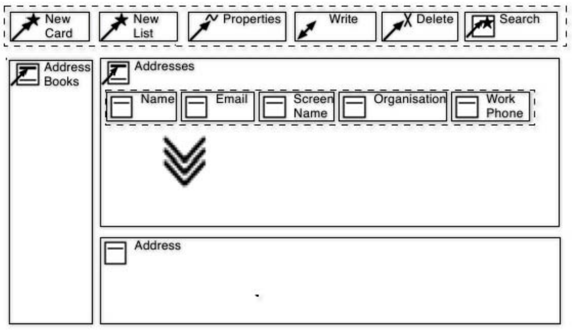

Figure 1: The top-level Canonical Abstract Prototype interaction space for Thunderbird's address book.

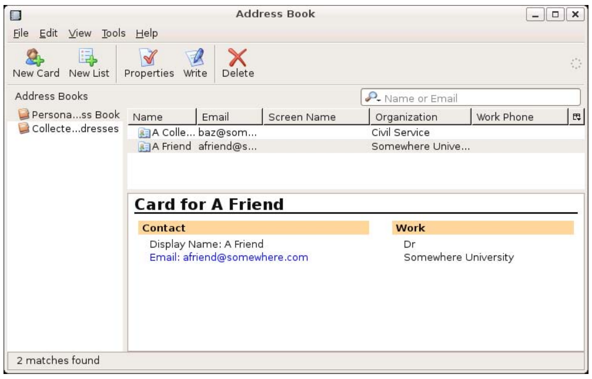

Figure 2: The top-level window for Thunderbird's address book.

2.2 Navigation Maps

Navigation maps describe the overview of all the interaction spaces, and the transitions that can occur between one interaction space and another. Lockwood and Constantine suggest some notation to describe particular types of interaction spaces, and certain categories of transitions. One useful feature of these navigation maps is that the individual elements within the map represent a context switch (i.e. transition) for the user. The principles of structure and visibility suggest that minimising the number of context switches while performing a task will make the task easier for the user.

3 Modeling Process

We will now discuss our proposed process for combining existing design models with event data extracted from the executing prototype.

The architectural design models are (semi-automatically) used in collaboration with event traces captured from test use to identify what parts of the architecture are actually used during a task, and how the used architectural components is used.

The overall process can be seen in figure 3. We shall discuss each of the six activities in order.

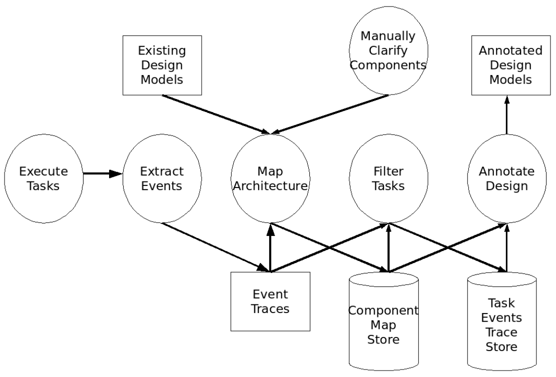

Figure 3: The process for converting event traces from actual user interaction into annotated design models that comment on the visibility and structure of tasks in the proposed architecture. The activities are represented as circles, the most important data as rectangles, and the major data stores as cylinders.

3.1 Execute Tasks

The first step is for users to use an executable prototype closely implementing the intended design. This use should be based around the key tasks that underpin the architectural model, and can be done "in the wild" outside of usability evaluation labs.

3.2 Extract Events

While the user is using the executable prototype, our process requires the non-intrusive extraction of event data. This data is combined into an *event trace*, and can aggregate information regarding the execution of different tasks, or even the same task execution multiple times.

The trace will contain information regarding the input device events, the update events that cause changes in the user interface, and the layout and nature of the implementation components while events are occurring. The trace also provides sufficient information to identify which implementation components a particular event occurred on, and when the event occurred.

Examples of input device events are pointer movements and button clicks, where the term button covers such things as a key on a keyboard or a button on a mouse.

3.3 Map Architecture

Once a trace has been extracted from the prototype, the implementation components need to be mapped to abstract components, and the containers in the implementation need to be mapped to the interaction spaces.

Our approach requires the storage of the original canonical abstract prototypes and navigation maps in a machine-readable format. We are currently developing tool support for generating these models, and we have developed an XML schema for the Canonical Abstract Prototype notation. However, it is certainly plausible that the formats used by existing tools, such as CanonSketch (Campos & Nunes 2005) could be utilised.

Using the existing (machine-readable) designs, the structural information in the traces is utilised to map the implementation components in the trace to the abstract component in the designs. Since the events are tied to the implementation components, this mapping enables the mapping of those events to the abstract components.

Once a mapping has been identified between implementation and abstract, then (assuming that the prototype's interface does not support a significant amount of layout configuration), this mapping should be largely reusable for later traces on the same interface.

An sample model for Thunderbird's address book can be seen in figure 4.

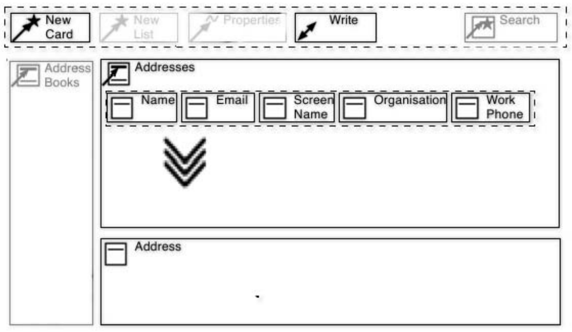

Figure 4: A revised Canonical Abstract Prototype of Thunderbird's top-level interaction space for the address book, indicating what abstract components were utilised (via degree of transparency) during a set of tasks completed by the user. An additional annotation could be to show how the input devices moved their focus from one component to the next.

3.4 Manually Clarify Components

It will be extremely difficult to correctly map all implementation components and containers to abstract components and interaction spaces. Therefore, there will be the need for the designer to be able to clarify any missed mappings, or correct erroneous mappings. The designer's manual clarifications feed into the Map Architecture activity.

3.5 Filter Tasks

Given that the event traces can contain events and implementation components for multiple tasks performed during one user session, it is useful to divide the traces up by task if we want to determine how a particular type of task uses the architecture. One approach to implementing a simplified version of this problem (that assumes that tasks do not overlap) would be to designate certain abstract components as being the sources and sinks of a cycle of interactions for particular tasks. This would require an extension of the Canonical Abstract Prototype notation.

3.6 Annotate Design

At the end of the process, our tool will then visualise the participation of each abstract component in a set of designer-selected task traces. For example, we can fade out those abstract components that are rarely used in the current set and increase the emphasis of those abstract components that are often used.

The aim of this is to identify:

1. Whether the architecture supports the structure of a task by seeing if the emphasised abstract components are limited to a few interaction spaces.
2. Whether the architecture supports the visibility of a task by not including a lot of unused abstract components in that task's key interaction spaces.

The event timings can also be used to indicate how the users moved within and between interaction spaces by showing the order in which components had events occur on them.

4 Challenges

There are a variety of technical challenges in our approach.

4.1 Identifying Abstract Components

It is very difficult to identify abstract components solely from the events and implementation components that are in a prototype's user interface. To achieve this identification, we need to use the existing design models to identify what an implementation component is supposed to represent.

Even matching implementation components in a user interface to abstract components in the associated design models is non-trivial however. Two reasons for this are the multiplicity and spatial relationships between the different levels of components.

Multiplicity Relationship

Abstract components may translate to multiple implementation components or, more rarely, the inverse may be true. This means that some pattern recognition needs to be done to identify common groupings of implementation components that frequently represent a particular abstract component.

Spatial Relationship

The naive approach would be to use exact positions and sizes of implementation components and abstract components to make matches. However — even leaving aside the possibility that the prototype is configurable enough that parts of the interface can be moved around — it is unlikely that the height/width ratio of the windows will be identical, or that the abstract and implementation components are exactly the same size or in exactly the same position,

4.2 Matching Interaction Spaces

Matching interaction spaces to the implementation containers in the prototype will — while non-trivial — be easier than mapping the abstract components to the implementation components. Implementation containers tend to be easier to map to an interaction space type purely based on the type name of the container (such as Java's *JTabbedPane*). This will especially be the case if the abstract components have already been correctly identified, since the input device events will then allow the right transition to be identified in the map.

5 Inquisitor

Inquisitor is our prototype tool that is currently being developed to support our process. Inquisitor works on Java programs. We will first discuss the event extraction mechanism we have implemented, and then move on to discuss the model annotation mechanism we are currently implementing.

5.1 Event Extraction

The first phase is to extract events from the executing prototype. Inquisitor uses a technique based on the *jRapture* tool to extract event information (Steven et al. 2000). Inquisitor acts as a wrapper for the prototype's main class, and registers itself with Java's AWT Toolkit class so that it can receive all AWT events generated during execution. Inquisitor also used the Java SPI libraries to implement various service providers that can then handle particular types of events.

The AWT events are received via the AWTEventListener interface that Inquisitor implements. Once Inquisitor has determined the type of event received, the event information is passed on to the appropriate service provider(s). One motivation for using the service providers is to easily support multiple extensible plug-ins consuming events, in a manner similar to that described in the *Strategy* and *Builder* design patterns (Gamma et al. 1994).

One of our services is a writer that outputs XML based on an XML schema that we have defined to represent the essential structure and events of generic GUIs. We consider the essential structure for each GUI component to include: the x and y location; the height and width; whether it is visible; whether it is enabled (i.e. accepting events); a unique identifier; a list of any sub-components (thereby recording the hierarchy of components on a per window basis); and any associated events.

We used JAXP and Apache Xerces's serialization capabilities to output the XML to the file system. To ensure serialization occurs only on shutdown we utilised the Java Runtime shutdown hook feature.

5.2 Model Annotation

The second phase is post execution analysis of GUI components and events. We do this post execution to minimise impact on the prototype's response times. Also, since we save the trace to disk during shutdown, we cannot perform heavy processing during shutdown as there is no guarantee it will complete before the JVM finishes.

One issue is that the capture trace files do not provide enough information to generate some models. For example, canonical abstract prototypes have the concept of tools which may have specific functions such as add or delete elements. This requires us knowing what each implementation component does and how the application logic reacts to events on the component. Other than the designer manually specifying this or using the location and sizes of abstract components in existing design models, there are a few possibilities available: Java annotations describing each component; and the AWT component name field. Unfortunately, neither can be assumed to be present in a prototype.

6 Related Work

Monitoring events in user interfaces is not new or novel. There has been significant work over the past fifteen years on automated analysis of user interface usage (Ivory & Hearst 2001). Many of these tools focus on analysing the specific interaction techniques (Guimbretiére et al. 2007), efficiency of tasks against some ideal execution, or summate total usage to identify certain interaction frequencies or errors (Hilbert & Redmiles 2000). Others have begun to do work on process validation, comparing event traces against behavioural models (Cook & Wolf 1997).

A number of event monitoring tools are also designed to work in collaboration with other data sources, such as video or audio recordings of the users in a usability lab (Weiler 1993). Our approach also aligns well with the industry accepted practice of gathering actual usage data to improve the design of GUIs as exemplified by the Microsoft Customer Experience Improvement Program (Microsoft 2007).

Our system — while similar — does differ in so far that the tasks are actually analysed to evaluate the visibility and structure of certain interaction spaces in the architecture. It is also novel in so far that we do not know of any other work that is directly applying these techniques to the modeling artifacts of usage-centered design.

7 Conclusion

In this paper we presented an automated process for comparing user interface design models to actual usage, using data collected from running prototypes of the designs.

Our process and tools will allow the UI designers to validate their models against widely used principles, and complements the more costly, intrusive and extensive user testing that requires manual observation.

In the future we hope to conduct a number of case studies demonstrating that our process provides the necessary feedback to improve a given design's support for the users and their tasks.

References

Campos, P. F. & Nunes, N. J. (2005), 'Canonsketch: A user-centered tool for canonical abstract prototyping', *Lecture Notes in Computer Science* **3425**, 146–163.

Constantine, L. L. (2003), Canonical abstract prototypes for abstract visual and interaction design, *in* J. Jorge, N. Nunes & J. F. e Cunha, eds, 'Proceedings of DSV - IS'2003 - 10th International Workshop on Design, Specification and Verification of Inter-active Systems', Constantine & Lockwood Ltd, LNCS - Lecture Notes in Computer Science, Springer-Verlag.

Constantine, L. L. & Lockwood, L. A. (1999), *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, Pearson Education.

Cook, J. E. & Wolf, A. L. (1997), Software process validation: quantitatively measuring the correspondence of a process to a model, Technical Report CU-CS-840-97, Department of Computer Science, University of Colorado.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Guimbretiére, F., Dixon, M. & Hinckley, K. (2007), Experiscope: an analysis tool for interaction data, *in* 'CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, New York, NY, USA, pp. 1333–1342.

Hilbert, D. M. & Redmiles, D. F. (2000), 'Extracting usability information from user interface events', *ACM Comput. Surv.* **32**(4), 384–421.

Ivory, M. Y. & Hearst, M. A. (2001), 'The state of the art in automating usability evaluation of user interfaces', *ACM Comput. Surv.* **33**(4), 470–516.

Microsoft (2007), 'Microsoft customer experience improvement program', `http://www.microsoft.com/products/ceip/en-us/default.mspx`.

Steven, J., Chandra, P., Fleck, B. & Podgurski, A. (2000), 'jrapture: A capture/replay tool for observation-based testing', *SIGSOFT Softw. Eng. Notes* **25**(5), 158–167.

Weiler, P. (1993), Software for the usability lab: a sampling of current tools, *in* 'Proceedings of INTERCHI 1993'.