IOWA STATE UNIVERSITY
Digital Repository

2011

Modeling error-based Adaptive User Interfaces

Karthik Narayanan Ramalingam

Iowa State University

Modeling error-based Adaptive User Interfaces

by

Karthik Narayanan Ramalingam

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Leslie Miller, Major Professor

Shashi K. Gadia

Sree Nilakanta

Iowa State University

Ames, Iowa

2011

DEDICATION

I would like to dedicate this thesis to Supreme God for the blessings and to my parents Ramalingam and Manimegalai for their unconditional support and encouragement throughout my work.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I take this opportunity to thank everyone who has helped me directly or indirectly throughout my Master's degree and during the course of my thesis.

Firstly, I would like to thank my advisor, Dr.Leslie Miller for helping me gain an insight into the research topic and guiding me in every aspect starting from the basics to the finer details of my thesis. Dr.Miller's constant encouragement, patience and support has been invaluable during the course of my Master's degree. I would like to thank Dr.Miller for his extreme patience during the learning phases of my Graduate study.

I would also like to thank my committee members - Dr.Shashi K.Gadia and Dr.Sree Nilakanta for taking time to review my thesis and giving suggestions for improvement. I would like to thank Iowa State University for providing us with state of the art facilities and resources to make ISU one of the best places to pursue higher studies.

I would like to thank all of my friends who stood by me during difficult times giving me the right mix of fun, advice and memories and a wonderful two years in Ames. Last but not the least, I would like to thank my parents for their continuous patience, understanding, advice, encouragement and support during my Graduate education. Their encouragement has played an inevitable role in pushing me to higher levels.

ABSTRACT

The advantage of bringing computation to field applications has led to an increase in location-based computing by many government and industrial organizations. Putting computing devices in the field creates a number of interesting challenges. Issues such as smaller screen size and weather complications are added to issues like varying levels of user ability. Adaptive interfaces provide the designer with the possibility of mitigating some of these challenges.

This thesis describes an error-based approach for adapting user interfaces to enhance performance in field settings. The proposed approach builds on vision and motor errors as a means of triggering the adaptive interface. The model and the current status of the approach are discussed.

CHAPTER 1.   OVERVIEW AND MOTIVATION

This chapter provides an introduction to the basics and the motivation behind the thesis. This also includes a brief overview on adaptive user interfaces and the overall structure of this thesis.

1.1   Introduction

An User Interface can relate to many types of interfaces to a computer. Any interface in which an user works on and interacts with fall under this category. The use of computation in field applications has increased. This in-turn has increased the number of location-based computing systems used. The usage of computing devices in the field provides with lots of challenges to deal with in extreme conditions like poor visibility, unexpected weather changes and many more. In such a situation the interaction of the user with the system can become very difficult. Adaptive interfaces are a way to counter these issues. In this thesis we discuss a model based on errors made by the user and the way for building an adaptive interface with the help of this model.

1.2   Adaptive User Interfaces and Models

An Adaptive User Interface is an user interface which adapts itself, changes its layout and rearranges the screen elements based on user needs. This makes the adaptive user interfaces useful in many areas. As a result, adaptive user interfaces form a formidable solution for many mobile applications. Mobile devices are an important area for making use of adaptive user interfaces. Mobile devices can support adaptive interfaces to meet the needs of different users. Adaptive interfaces can also be useful in aircraft displays and other military devices. Having

listed its uses we must also note that there will be a substantial increase in the complexity of designing such interfaces.

The construction of adaptive user interfaces can be streamlined by the use of models, which form a standard for creating these interfaces. There are several types of models capable of accommodating individual user requirements. This helps us to study and understand the behavior of users which later can be extended into the adaptive interfaces.

1.3 Computation in extreme conditions

As mentioned earlier, the use of computers has been extended to many areas. This in turn increases the amount of usage of user interfaces in those applications. By extreme conditions we refer to situations like a farm field during a severe weather, cockpit of a flight flying in a bad weather, military field applications, training operations and many more. Having listed them, we are aware that usage of technology in such areas is very predominant. Such environments can be highly difficult to work in. The users are susceptible to errors and hence the need for adaptive user interfaces becomes vital. While the techniques that we propose could be used in any environment, we have focused on field applications due to their susceptibility to external conditions.

1.4 Organization of the Thesis

The thesis focuses on the models developed and details the implementation of those models. We present two models of an adaptive user interface based on user errors and provide a methodology for building the model from an existing interface design. Later we discuss about the implementation of those models. The implementation makes use of XML representations of the individual user interface screens. Our current implementation is based on Java, but this could be extended to other environments.

The chapters in this thesis are organized as follows:

Chapter 2 discusses the previous work done in adaptive user interfaces. It also discusses the use of models in building adaptive interfaces along with examples of work involved using

these models.

Chapter 3 describes the two models that have been developed - error model and interface model presented by us. It details the construction and usage of those models.

Chapter 4 looks at the implementation of the adaptive user interface based on the models described earlier in detail.

Chapter 5 wraps up the thesis by discussing the work and the future work which can be done to improve the models and adaptive user interface.

CHAPTER 2.   REVIEW OF LITERATURE

Adaptive User Interfaces refers to a very broad category of interfaces. It can be precisely defined as a user interface which has the ability to adapt and change based on the user's performance. The interface is responsible for learning about the user differences and preferences in order to make the decision for the user. This can be achieved by constructing an user model. This chapter will examine the various approaches that have been proposed for modeling adaptive user interfaces. Further, the various challenges involved in developing the models for adaptive user interfaces will also be examined.

Several authors have looked at the possible ways for accommodating individual differences. In their paper Buring *et al.* (2006) analyzed the usability of user interface with respect to individual differences in spatial ability by using an interface which support zoom with overview and detail. They found out that the users were able to perform much faster using an interface which had detail rather than the one with an overview.

Vicente *et al.* (1988) conducted an experiment to compare the performance of users with high and low spatial abilities. They were able to overcome the poor performance due to low spatial ability by making some small changes to the interface like addition of extra commands. One important point to be considered is how far can we improve the usability of the systems? Each and every individual has a different set of cognitive skills and preferences. Benyon (1993) has differentiated the individual differences that are useful, those which are stable and have an impact on interaction. Stanney and Salvendy (1995) were successful in using visual mediators to accommodate low spatial individuals. Their experiment found it to be successful to improve the search performance of low spatial individuals.

Newell and Gregor (2000) suggested the development of a new inclusive design which includes people with disabilities. Keates *et al.* (2002) presented a methodological design approach

for implementing inclusive interface design. Pattison and Stedmon (2006) have discussed the use of an inclusive interface for cell phones concentrating on the usability issues related to older users. Krish (2009) in his paper about adaptive user interfaces for health-care systems discusses about the current knowledge, goals, and other significant characteristics of the user which are important for redesigned interfaces.

As seen above though high amount of work has been done on inclusive design approach, it should be noted that in an inclusive interface all the users are forced to use the same interface. This may not be the right solution. Interfaces appearing easier to use by a certain group of people may not be easier for a large number of people. Benyon (1993) has pointed out that such an interface would result in non-optimal interface for certain users.

Thus to provide a much more accommodating solution, user interfaces must possess the ability to adjust and adapt to individual user preferences or differences. There are wide varieties of device types, form factors and input methods which make it highly difficult for the programmers to create an interface which accommodates all these differences hence an automated adaptive system becomes necessary under these circumstances. It must be noted that there is a key difference between the interface being adaptable and adaptive. Adaptable interfaces are those in which the users are given a choice. These interfaces are always in user's control, but not all adaptive interfaces are controlled by the users.

In their study Findlater and McGrenere (2004) found out that users responded well to adaptable and adaptive user interfaces over the static interfaces. Adaptive interfaces can be differentiated from adaptable interfaces by the means of their overall performance and details in implementation. Adaptive interfaces require extra overhead in implementation.

Modeling user behavior is an interesting area as it will help us with new insights on the nature of human interaction with systems. It standardizes the way of building an adaptive user interface. It is vital to create a model for the effective implementation of adaptive user interfaces. Kules (2000) has discussed several techniques of user modeling and adaptive systems. The paper also provides a set of guidelines in building an adaptive user interface, in which he has emphasized user modeling. In their design of an adaptive route advisor Rogers *et.al.* (2000) have used a model of driver preferences. The route advisor constantly updates the user model

by interacting with the user and gathering user preferences. In developing the personalized word assistant based on episodes identification and association, Liu *et al.* (2003) have recognized user behavior patterns and built a user profile that facilitates personalized interactions.

The most prevalent common example of adaptive user interfaces would correspond to those systems which are used to filter information and recommend users accordingly. Content based and collaborative filtering are among the basic approaches involved in adaptive systems. Balabanovic (1998) discuss a system which retains profiles for individuals and topics later combines their predictions to produce both content-based and collaborative behavior. Pat Langley (1997) discusses problems which involve more than just selecting from among a large set of documents or products. Horvitz et al. (1998) built statistical methods and cost-benefit approaches to identify decisions on informing users in the area of context-aware interfaces and environments.

Gajos et al. (2006) found that increasing the accuracy of the adaptive system significantly improved both performance and adaptive interface utilization. Further, both predictability and accuracy significantly increased participants' satisfaction. Shankar *et al.* (2007) showcases statistically significant results which showed that an adaptive context-aware user interface can improve user-experience.

An adaptive interface is rated based on the effectiveness of the algorithms which are used to determine the differences and preferences of the users. Langley (1997) has suggested the use of machine learning algorithms for developing such interfaces. The adaptive interfaces must be able to build suggestion models which provide only recommendations to the user through the collective learning of user preferences/differences. It should be noted that the knowledge gained by the adaptive system must be capable of reflecting the preferences learnt.

Viano *et al.* (2004) developed an adaptive interface that focuses on the state of the process and state of the user. Tsandilas and Shraefel (2004) have examined the accuracy of algorithms for predicting user performance and satisfaction. Taylor *et al.* (2009) investigated the use of user error detection as a means of adapting web pages to suit the abilities of older adults.

This thesis aims at modeling an adaptive user interface based on the errors made by the users in the process of interacting with the system. It's intuitive that errors made by a user can be easily corresponded to adapting the system such that it minimizes the errors committed by

the user. The system built is capable of indexing the components on the interface, capturing the errors made by the user and produce multiple interfaces based on user preferences and/or errors made by the user.

CHAPTER 3.   MODELS

This chapter describes the models based on which the adaptive user interfaces are built.

3.1   Error Model

A user is likely to make errors while using an interface, especially in adverse conditions. These user errors occur for many reasons. Some of them are:

- Environmental conditions (eg. Low visibility)

- User limitations (eg. Poor motor skills)

- Complexity of the interface

Our model focuses on tapping errors such as a user missing a button while tapping, incorrect taps where the user taps a wrong button, reversals and text entry errors. It is not easy to identify the reasons behind these types of errors. The reason for a missed tap can be anything from a user not being able to locate the button to the user not being able to hit the button properly. Figure 3.1 lists the types of errors supported by this model.

The two types of tapping errors supported are reversals and missed taps. A reversal refers to the act of a user where he/she taps on a component and "quickly" reverses the action. The model supports missed taps by maintaining a threshold value for the distance around each component. If the missed tap is within the range of the threshold of a particular component then that tap is considered as a missed tap for that component.

The taps which do not fall under the radius of any of the components threshold value are considered to be errant taps. A series of errant taps suggests that the user might have difficulty in accessing the right component. Thus by capturing repeated errant taps we can increase the threshold value for that particular component to capture the missed taps.

Tapping Errors
Reversals
Missed Taps
Check Box
Radio Button
Menu
Text Box
Text Area
Button
Keyboard

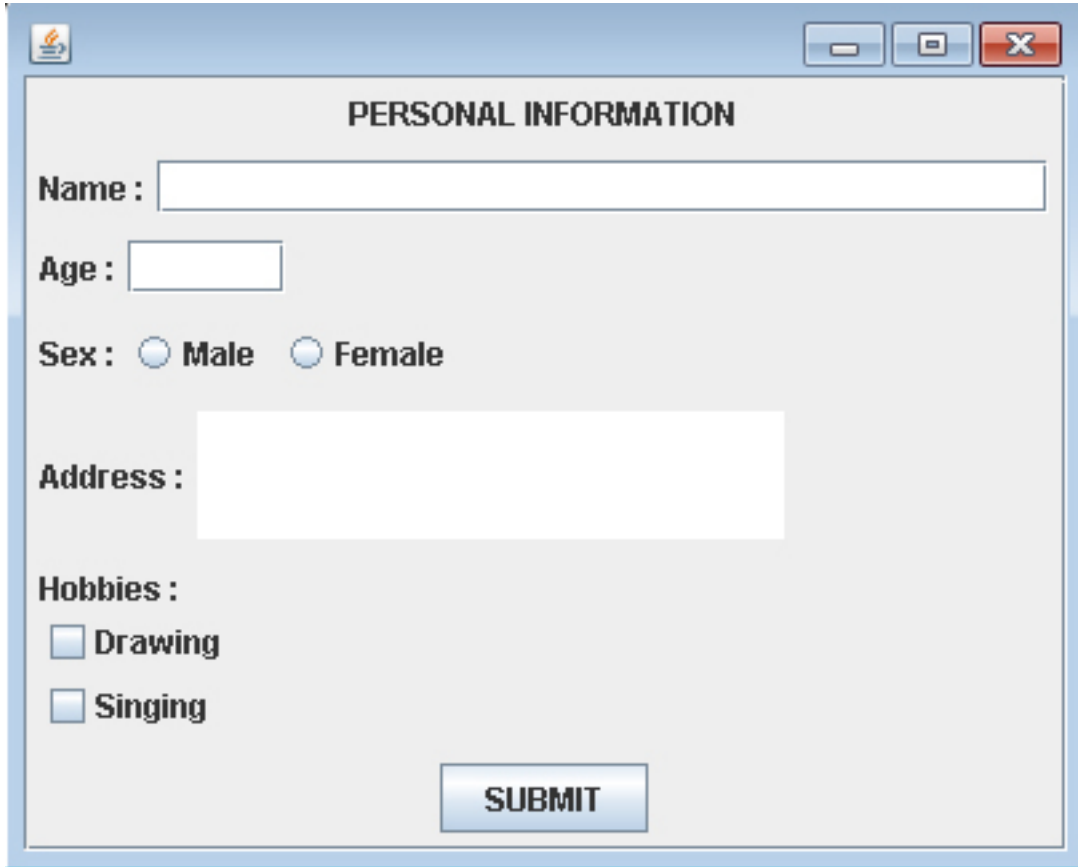Figure 3.1   Type of errors.

3.2   Interfaces

3.2.1   XML Representation

The use of an XML representation is very useful in creating the automatic error correction code. This also helps us to build an index of components. The benefits of indexing components come into picture when there are a large number of components present in the interface. The indexing helps us speed up the process of using our error model. The process of indexing and its uses are explained more in detail in the implementation section.

For building the xml representation all we need is the physical location of the components in the screen/layout. All the components which we include in the interface are physically represented by a bounding box. It should be noted that the physical location of the components refers to the location of the components with respect to the layout and not the screen of the device. Usually, the location of the components will be related to the location which appears on the screen of the device. In reality, there might be situations where the size of the screen built for an application can be much larger than the original size of the device's screen. Hence, to accommodate all types of screen size the physical location of the components must be relative to the screen of the interface, as opposed to the device.

Components like label and description text which are trivial must also be considered while

making an entry about the location of non-trivial items like textboxes, radio buttons and other such components in to the XML representation. This gives us the exact representation of the whole interface, which is very useful for the external model discussed below to create the code automatically.

Figure 3.2   A sample user interface.

The component's name and its location form the essence of the xml representation. The location of the components are identified by the corresponding x,y co-ordinates of the components top left corner and bottom right corner both with respect to the outermost frame, this also gives us the measure of the components size. We also have the information about the overall size of the interface. A sample user interface and its corresponding XML representation are shown in Figures 3.2 and 3.3, respectively.

```
- <Screen screenId="1">
  - <ScreenSize>
      <Width>407</Width>
      <Height>329</Height>
    </ScreenSize>
  - <Components>
    + <Panels>
    + <ScrollBars>
      <Labels />
    - <TextBoxes>
      - <nameTxt>
        - <Start>
            <X1>49</X1>
            <Y1>31</Y1>
          </Start>
        - <End>
            <X2>383</X2>
            <Y2>51</Y2>
          </End>
        </nameTxt>
      + <ageTxt>
      </TextBoxes>
    + <TextAreas>
      <CheckBoxes />
      <RadioButtons />
    + <Buttons>
      </Components>
    </Screen>
```

Figure 3.3   Fragment of XML Representation for screen shown in Figure 3.2.

3.2.2 External Interface Model

The user interface of an existing application consists of a set of screens. Each screen has a set of subtasks (T) associated with it. In our current model, we define two types of tasks: transition tasks and enhancement tasks. *Transition tasks* complete a task for current screen and transfer control to a new screen. *Enhancement tasks* either change the structure or add data to a component of the current screen, but the user continues to process the same screen content. For each t ϵ T, there is a set of components.

Let s be a screen from an existing application that has k tasks assigned to it. If s has more than one task assigned to it, we allow the introduction of new screens that only provide a proper subset of the tasks assigned to s. We use the term *error-based screens* to indicate these new screens. For each s and the set of error-based screens that can be generated from it, we define an *action tree for screen s* to be a labeled, directed tree that has s as its root node, screens with a single task (or stand alone tasks) assigned as its leaves, and intermediate nodes (screens) have at least one less task than their parents and at least one more task than any of its children. Moreover, any tasks assigned to a node must have also been assigned to its parent. Nodes are labeled with the task type (transition or enhancement) and edges are labeled with a conjunction of equalities of the form 'error type=count', where count is the number of errors of this type that cause transition to the error-based screen.
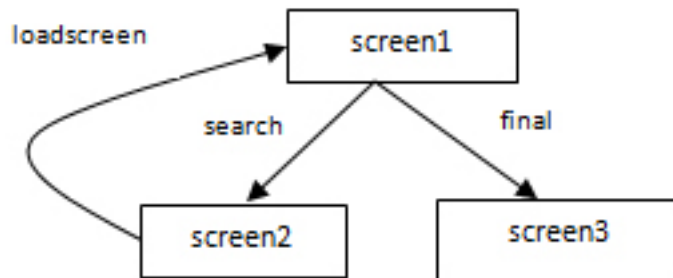

Figure 3.4   Interface graph for a simple three screen user interface.

An *interface graph* is a labeled, directed graph G = {(N,E),L}, where N is the set of screens created for an application. Figure 3.4 shows an interface graph for an application using three

screens. We omit the XML label due to space limitations. The screens in N are of two types, namely, screens from the original application and error-based screens. L is the set of directed edges (n1, n2) such that n1, n2 ϵ N and a user can move from screen n1 to screen n2 either by completing a task on n1 or by reaching an error threshold on n1.

The labels are represented by L = L1 $\bigcup$ L2, where L1 is the set of node labels and L2 is the set of edge labels. Each label l ϵ L1 consists of the XML representation of the screen as defined in the previous subsection and the node type (existing, transition, or enhancement). Labels in L2 are literals expressions consisting of the conjunction of the 'error type=count' raised on n1, or the name of the completed task.
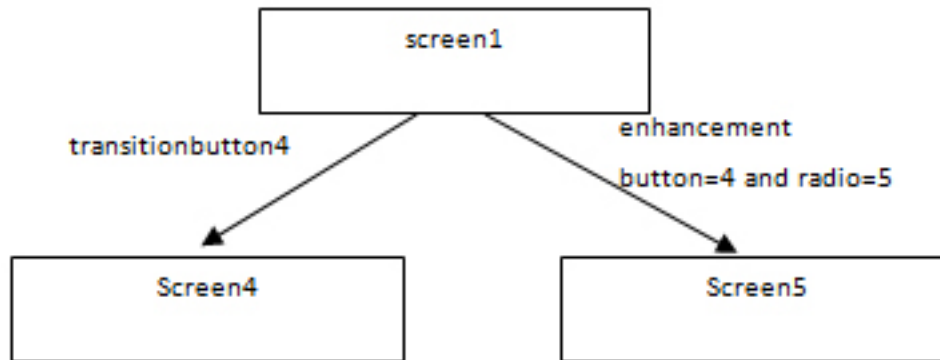
Figure 3.5   Sample action tree for screen1 in Figure 3.4.

Once we have the task set(s), the task set types, and the error types/counts, we can generate the action tree for each screen in the original user interface. Figure 3.5 illustrates a sample action tree for screen1 (Figure 3.4). The example assumes that screen1 has two tasks that can be identified, namely, a transition task and an enhancement task. The action tree can be integrated with the original interface graph to form the interface graph for the error-based user interface (Figure 3.6). The resulting interface graph provides the information needed to automatically generate the layout portion and the error capture and analysis portion.

Figure 3.6 Interface graph resulting from the integration of the action tree with transition/enhancement labels omitted.

The implementation of these models is discussed in Chapter 4.

CHAPTER 4. IMPLEMENTATION

In this chapter details of the implementation of the models are discussed. Java and Java Swing has been mainly used for the interface implementation, along with XML and the DOM parser for the XML representation of the interface. F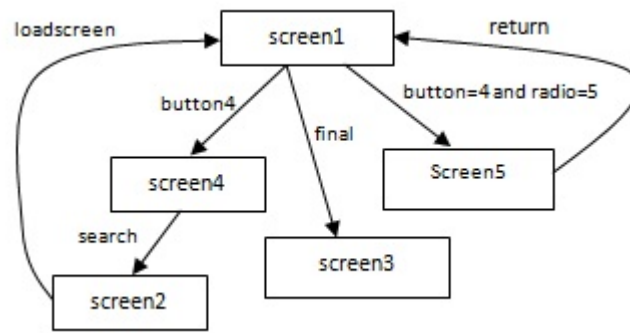igure 4.1 shows a sample screen. It will form a running example on which all the implementation details will be discussed in detail. The sample screen is a simple interface used to collect the personal information of a candidate. Once the user fills in all the information and clicks the submit button the information will be stored in the user database. The interface makes use of basic java swing components like labels, textboxes, text area, checkboxes, radio buttons and command button.

4.1   Implementation of the XML representation

Let us look at the implementation of the XML representation of the interface. The goal of this module is to create a XML file which represents the components and the user interface itself. Hence, for this purpose the name of the components, size of the components and their corresponding location are needed. This XML representation will be useful in situations where we need to recreate the interface as a whole or in part. The XML representation also aids us in indexing the components based on their location.

As shown in the Figure 4.1 the screen consists of various swing components ranging from labels to buttons. All of these components must be present in the XML representation of the interface. In Java, DOM class can be used to create a XML file. Initially, an empty Document is created with the *DocumentBuilder class*. Next step involves defining all the XML contents with *Element class*. Finally, the *Transformer class* is used to output all the XML content in a File.

Figure 4.1 Sample Interface

Each of the swing components like labels, textboxes, panels and many more are defined as separate elements. Using the *container class* the whole interface is passed into a loop as a container of Java swing elements. In the successive iteration the component types are identified using the *instanceof* operator. The name, size and location of the component are also retrieved and added to the element. It is imperative to note that the components must have a valid name in order to create the xml representation. In the case of an unnamed component it will be named with an arbitrary name which can be easily related to the component.

A component is located by its top-left corner position. This is embedded within the *start* XML tag. The *end* tag has the bottom-right corner position of the component. Thus by this way both the *start* and *end* tag collectively represents the size and location of a component. A sample code snippet used to add the JButton component in the XML representation is presented below and its corresponding XML entry is showed in Figure 4.2.

```java
if(child instanceof JButton)

{

if(child.getName()!=null)

{

Element button = doc.createElement(child.getName());

Element start = doc.createElement("Start");

Element x1 = doc.createElement("X1");

Element y1 = doc.createElement("Y1");

Element end = doc.createElement("End");

Element x2 = doc.createElement("Y2");

Element y2 = doc.createElement("Y2");

x1.appendChild(doc.createTextNode(Integer.toString

(components.get(child.getName()).x)));

y1.appendChild(doc.createTextNode(Integer.toString

(components.get(child.getName()).y)));

x2.appendChild(doc.createTextNode(Integer.toString

(components.get(child.getName()).x+child.getWidth())));

y2.appendChild(doc.createTextNode(Integer.toString

(components.get(child.getName()).y+child.getHeight())));

start.appendChild(x1);

start.appendChild(y1);

end.appendChild(x2);

end.appendChild(y2);

button.appendChild(start);

button.appendChild(end);

buttons.appendChild(button);

}

}
```

```
- <Buttons>
    - <submitBtn>
        - <Start>
            <X1>155</X1>
            <Y1>257</Y1>
        </Start>
        - <End>
            <X2>233</X2>
            <Y2>283</Y2>
        </End>
    </submitBtn>
</Buttons>
```

Figure 4.2   XML representation of the Submit button present in Figure 4.1.

4.2   Implementation of Components Index

Implementing an index for the components present in the interface helps reducing the search time and comparison time of the components in developing the error model. We have the necessary details for creating an index in the xml representation of the interface.

Figure 4.3   Scenario illustrating the inefficiency of single-point indexing

Indexing the components based on a point on the component would be a proper fit. For this purpose we can use the centroid of the component or any one of the corner of the component. But such an idea would not suit the best in all scenarios. One such scenario where this idea would not work is illustrated in the Figure 4.3. As shown in the figure, there are two components located close to each other but the sizes of these components differ by a large margin. Visually it is clear that the tap is intended for the 'Name' textbox. Let us consider that either centroid of the components or any one corner of the components (marked by a dot) is used for indexing. In this scenario, once the location of the missed tap is compared with the closest distance in the index it will yield the 'Age' textbox as the result. Hence, the concept of single point

indexing will mislead the detection of missed taps. On the other hand, having too many points for indexing increases the complexity of the implementation.

Figure 4.4   Top-left and Bottom-right corners of a component

The key to this is that the boundary of the component must be considered while implementing the index. Each specified boundary returns only one component. This leaves us with the option of considering the top-left corner and bottom-right corner (Figure 4.4) of a particular component. Any component represented using Java-Swing has a rectangular boundary for it. Even while considering a component like a radio button, it is bounded by a rectangular area to accept input (Figure 4.5).

Figure 4.5   Rectangular boundary around components

By choosing both top-left corner and bottom-right corner of a component it is guaranteed that all physical features of that component like height, width and location can be taken into consideration. As we need to consider the boundary of the components for indexing, a two level indexing is maintained. The first level contains the 'x' coordinates of the points and the second level contains the 'y' coordinates of the points.

The data type used for the indexing is *HashMap*. A sample definition of an index is shown below.

```
Map indexX = new HashMap<Integer, ArrayList<String>>();
Map indexY = new HashMap<Integer, ArrayList<String>>();
```

As shown, the index is a hash map of Integer and an array of Strings. The integer value corresponds to the 'x' or 'y' value of the component's top-left corner or bottom-right corner.

The array holds the name of the component which has one of its corners in the position indicated by the integer.

Figure 4.6 shows a sample of the index created for the screen shown in Figure 4.1. The usage of these indices to narrow down the components to be compared for the error model is explained in the next section.

X-Index

5	drawHob, singHob
38	ageText, maleBtn
49	nameTxt
64	addTarea
72	singHob

Y-Index

31	nameTxt
51	nameTxt
61	ageText
81	ageText
91	femaleBtn

Figure 4.6   Sample of Index created for the screen shown in Figure 4.1.

4.3   Implementation of Error model

The error model implemented makes use of the XML representation and the indexing of components discussed in the previous section. This model identifies the errors committed by the user while using the interface. By the term error it refers to those taps/clicks of the user which takes place outside the area of any of the components used in the interface. A threshold distance is maintained around each of the click-able components, such that if the tap lies within that range then it is considered as a *missed tap* corresponding to that particular component. If the tap is not inside any of the boundary then it is considered to be as an *errant tap*.

A *mouse listener* is added to the screen/interface. This helps us to retrieve the mouse click event for this interface. The value of the mouse click obtained is relative to the JFrame screen. Notice that mouse clicks and taps are the same event to the mouse listener. With the help of the index created for the components, an array of components to be compared is narrowed down. As there are two levels of index, one for x-axis and another for y-axis, the 'x' and 'y'
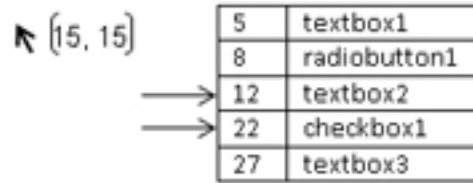
Figure 4.7   Components selection from index

coordinates of the mouse click is retrieved. These values are run through the indices to obtain the components to be compared for which the adaptive code has to be applied. While browsing through the index two values are retrieved, one greater and another lesser than the recorded click/tap (Figure 4.7). This in-turn returns the list of components which should be compared with the tap.

As indicated above the idea is to obtain two values from the index. Two flags are maintained, one for finding the immediate lesser value and another to find the immediate higher value. A fragment of the code used for finding out the components to compare is shown below.

```
for(int key: indexX.keySet())

    {

    if(key<me.getPoint().x && key>smallKey)

    smallKey=key;


    if(key>me.getPoint().x && key<largeKey)

    largeKey=key;

    }


    if(smallKey==-1)

    {

    int count=0;

    for(int key: indexX.keySet())

    {
```

```
if(count==0)

prevKey=key;

if(key<prevKey)

{

smallKey=key;

prevKey=key;

}

count++;

}

prevKey=-1;

}


if(largeKey==5000)

{

for(int key: indexX.keySet())

{

if(key<=smallKey && key>prevKey)

{

largeKey=key;

prevKey=key;

}

}

prevKey=-1;

}



for(String component : indexX.get(smallKey))

{

componentsToCompare.add(component);
```

```
}

for(String component : indexX.get(largeKey))

{

componentsToCompare.add(component);

}
```

This index helps us to reduce the components to be compared to a maximum of four components at any time. We end up choosing two entries from x-index and two from y-index. In some situations the components to be compared can even be lesser than four. There might be situation where both the entries retrieved from an index correspond to a same component. In the event of not using an index all the components present in the interface has to be compared which will reduce the efficiency by increasing the processing time to a greater extent. Thus a considerable amount of computation time is reduced. The next step is to find which of those selected components is nearer to the click/tap made by the user. If this distance lies inside the threshold value then the tap is considered as a missed tap pertaining to this component.

Figure 4.8   Example of a missed tap

The process of determining the closest component to the tap is not as simple as it sounds. Here we need to find which point(s) in the component has to be compared with for finding the closest distance from the tap. The boundary outside the component is taken into consideration which indicates the threshold distance, and if the point/tap is present inside this boundary we determine that the missed tap corresponds to that component (Figure 4.8 and 4.9). If there is more than one boundary inside which the tap falls in then we consider four corners and the midpoints of each axis of the components to compare the distance and determine the component which was missed.

If the tap is a missed tap the interface can be made to behave adaptive to the user's action

Figure 4.9   Example of an errant tap

by changing the size of the component for which the missed tap was recorded. There is lots of potential use by this model which is further discussed in the future works.

4.4   Implementation of External Interface Model

The external model is a separate model operating individually from the other models discussed previously. This model helps us to create multiple interfaces from a single interface. In this section, we will discuss about the implementation of software that breaks up interface screens, which will aid the process of implementing the external model. This software takes a particular interface as its input and produces several sub interfaces. Currently, a designer manually determines how existing screens can be broken up. The selection window of this model is shown in Figure 4.10.
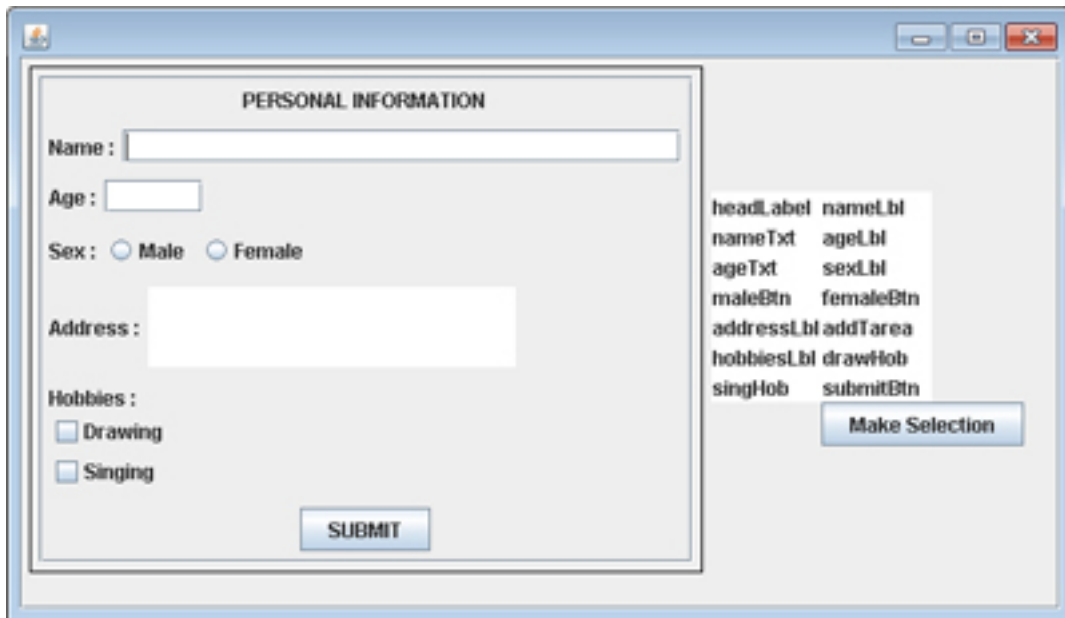

Figure 4.10   Selection window to break up the interface into sub interfaces.

As shown in Figure 4.10, the interface is embedded in a frame adjacent to the list of components present in the interface. The designer can choose any number of components from the list so that a new interface is created with the selected components. Once the components are selected, the component name and its size are obtained from the xml representation along with its location in the original screen. Using the details obtained, java code is generated automatically for the new screen which contains the selected components.

The first step involved is to initialize the selection window. The outermost JFrame of the original interface is obtained. This JFrame is run through a loop to obtain its children in all the levels. The code fragment used for obtaining all the child components is presented below.

```
public void loop(Container c)

{

for(Component child : c.getComponents())

{


if(child instanceof JTextField)

{

if(child.getName()!=null)

listModel.addElement(child.getName());


}

else if(child instanceof JLabel)

{

if(child.getName()!=null)

listModel.addElement(child.getName());

}

else if(child instanceof JRadioButton)

{

if(child.getName()!=null)
```

```
listModel.addElement(child.getName());

}

else if(child instanceof JTextArea)

{

if(child.getName()!=null)

listModel.addElement(child.getName());

}

else if(child instanceof JCheckBox)

{

if(child.getName()!=null)

listModel.addElement(child.getName());

}

else if(child instanceof JButton)

{

if(child.getName()!=null)

listModel.addElement(child.getName());

}

else if(child instanceof JPanel)

{

loop(((Container) child));

}

else if(child instanceof Container)

{

loop(((Container) child));

}

}

}
```

After obtaining the details of all the children, a selection list is created which contains the name of the components. A new screen is created which acts as the selection window. This screen contains both the original interface and the selection list. A total of three JPanels are used. One is the panel obtained from the original JFrame. Next panel contains the selection list that lists the names of the components present in the first panel. The above two panels are embedded in a single panel and is presented in a JFrame as a selection interface. The designer chooses the components which need to be presented in a new interface. After the creation of a new sub interface the components are removed from both the old interface as well as from the selection list (Figure 4.11).
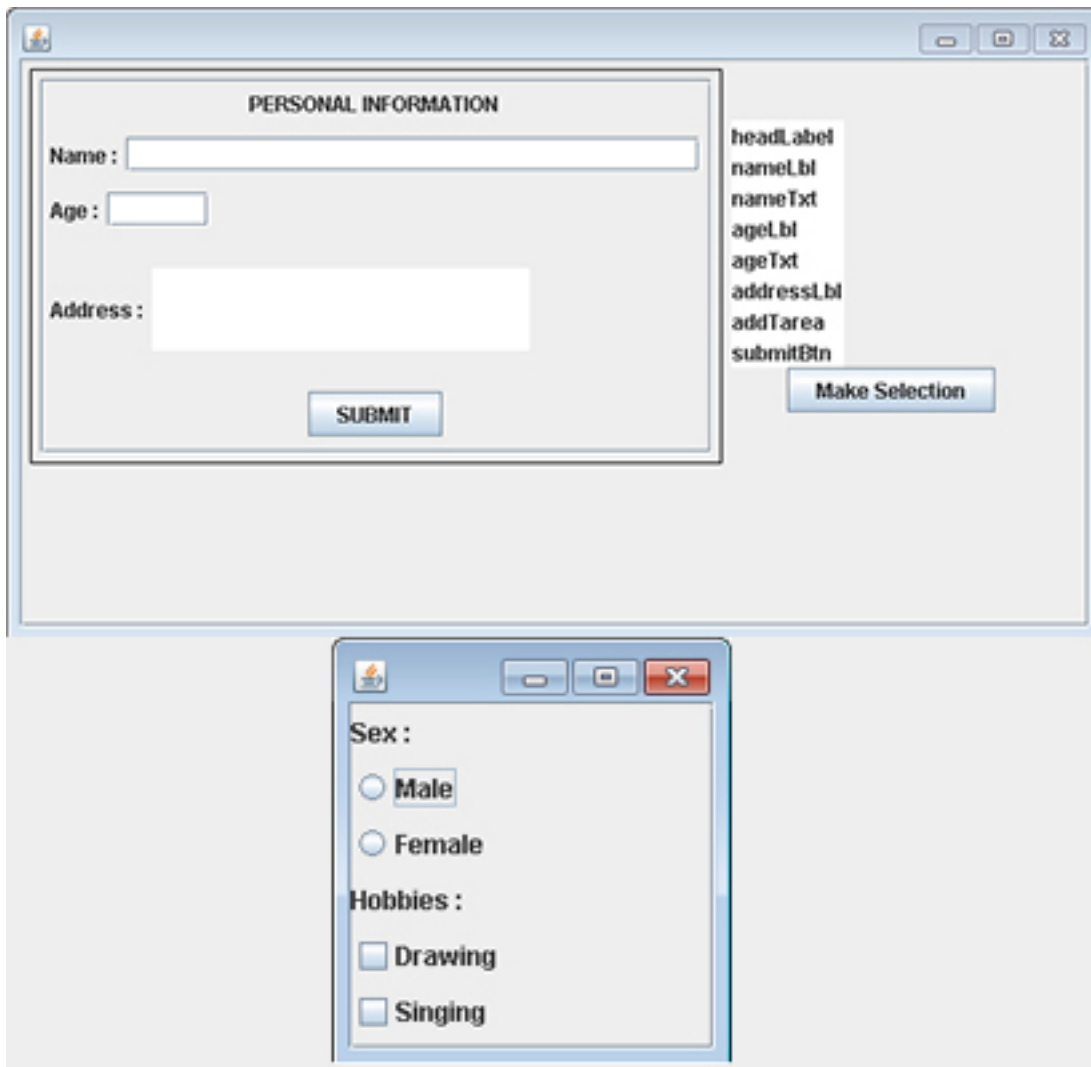

Figure 4.11   Creation of sub interfaces from an existing interface.

An important thing to be noted is that in the creation of new interfaces the components that are selected are not created from the scratch for the new interface. Those components are retrieved directly from the original interface as a Java object and are added to the new interface. Hence, the properties of those objects remain unchanged. For example let us assume that a particular component has a mouse listener event. During the construction of the new screen, the mouse listener event is also transferred along with the component.

This external interface model can be used in tandem with the error model described earlier. If a user commits multiple errors then the error model can be extended to display the selection model to create multiple new screens.

CHAPTER 5.   CONCLUSION AND FUTURE WORKS

This thesis described the user interface models developed for location-based applications. The reason for special emphasis on location-based applications is the susceptibility of the user interfaces in adverse conditions. We have focused on errors made by the users as a basis of forming the model. Later this can even be extended to user patterns which can support the model and also increase the error types supported by the model.

A sample application was created in which the two models were implemented and has been presented. These models can be added to any existing Java applications which process the interface, capture the errors and produces new screens automatically based on users selection.

Users are prone to commit errors while using the interfaces in location-based applications on fields in adverse conditions. The model developed can be tested more accurately in a real time setting where adverse weather conditions exist. It is not easy to test under these scenarios. Hence, virtual reality might be one of the best ways to test the stability of this model and can give helpful insights on how to improve the model. We can create the virtual environment of severe weather conditions or conditions where the visibility is less and hence we can test the accuracy of this model.

Apart from location-based applications these models can also be extended to other applications such as mobile devices and other handheld devices where user interfaces plays a vital role in communicating with the users. The idea of creating a model by itself helps to study the user preferences and will aid in improving the design of the interface.

BIBLIOGRAPHY

Balabanovic, M. (1998). Exploring Versus Exploiting when Learning User Models for Text Recommendation *User Modelling And User-Adapted interaction. 8(1-2).* 71-102.

Benyon, D. (1993). Accommodating individual differences through an adaptive user interface. In M. Schneider-Hufschmidt, T. Kuhme, and U. Malinowski (eds.). *Adaptive User Interfaces Results and Prospects. Amsterdam, North-Holland: Elsevier Science Publications.*

Buring, T., Gerken, J., and Reiterer, H. (2006). Usability of overview-supported zooming on small screens with regard to individual differences in spatial ability. *In Proceedings of the Working Conference on Advanced Visual interfaces (Venezia, Italy, May 23 - 26, 2006).*

Findlater, L. and J. McGrenere. (2004). A comparison of static, adaptive, and adaptable menus. *ACM Conference on Human Factors in Computing Systems. Vienna, Austria.* 89-96.

Gajos, K. Z., M. Czerwinski, D. S Tan, and D. S. Weld. (2006). Exploring the design space for adaptive graphical user interfaces. *Proceedings of the working conference on advanced visual interfaces. Venezia, Italy.* 201-208.

Horvitz, E., J. Breese, D. Heckerman, D. Hovel and K. Rommelse. (1998). The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users, *Proceedings of 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI 98).* 256-265.

Kay M. Stanney and Gavriel Salvendy (1995). Information visualization; assisting low spatial individuals with information access tasks through the use of visual mediators. *Ergonomics, Volume 38, Issue 6,* 1184 - 1198

Keates, S., P. J. Clarkson, and P. Robinson. (2002). Developing a practical inclusive interface design approach. *Interacting with Computers. 14(4).* 271-299.

Krish Ramachandran. (2009). Adaptive user interfaces for health care applications *Available at http://ibm.com/developerWorks*

Kules, B. (2000). User Modeling for Adaptive and Adaptable Software Systems. *Available at http://otal.umd.edu/UUGuide/wmk/.*

Langley, P. (1997). User Modeling in Adaptive Interfaces. *Proceedings of Seventh International Conference on User Modeling. Banff, Alberta.*

Liu, J., C. K. Wong, and K. K. Hui. (2003). An Adaptive User Interface Based on Personalized Learning *IEEE Journal on Intelligent Systems.* 52-57.

Newell, A. F. and P. Gregor. (2000). User densitive inclusive design - in search of a new paradigm. *ACM Conference on Universal Usability. Arlington, Virginia.* 39-44.

Pattison, M. and A. Stedmon. (2006). Inclusive design and human factors: designing mobile phones for older users. *PsychNology Journal. 4(3).* 267-284.

Rogers, S., C. N. Fiechter, and C. Thompson. (2000). Adaptive User Interfaces for Automotive Environments. *Proceedings of the IEEE Intelligent Vehicles Symposium 2000*

Shankar, A., S. J. Louis, S. Dascalu, L. J. Hayes and R. Houmanfar. (2007). User-Context for Adaptive User Interfaces *Proceedings of the 12th international conference on Intelligent user interfaces. Honolulu, Hawaii, USA.*

Taylor, A., L. Miller, S. Nilakanta, J. Sander, S. Mitra, A. Sharda, B.Chama. (2009). Using an Error Detection Strategy for Improving Web Accessibility for Older Adults. *ACHI 2009.* 375-380.

Taylor, A., L. Miller, S. Nilakanta. (2009). Comparing Strategies for Evaluating Older Adults to Improve Web Accessibility. *CATA 2009.*

Tsandilas, T. and M. C. Shraefel. (2004). Usable adaptive hypermedia systems. *New Review of HyperMedia and Multimedia. 10(1).* 5-29.

Viano, G., J. Alty, I. Angulo, D. Biglino, m. Crampes, V. Daurensan, A. Parodi, C. Khalil, C. Vaudry, and P. Lachaud. (2004). Adaptive user interface for process control based on multi-agent approach. *AVI 2004. Palermo, Italy.* 201-204.

Vicente, K. J., and R. C. Williges. (1988). Accommodating individual differences in searching a hierarchical file system. *International Journal of man Machine Studies, 29.* 647-668.