CS211 Lecture: The User Interface

Last revised November 15, 2004

Objectives:

1. To introduce the broad field of user interface design
2. To introduce the concept of User Centered Design
3. To introduce a process for user interface design
4. To discuss the difference between java awt and swing
5. To discuss layout options, including the GridBagLayout class

Materials:

1. Transparency of Braude figures 3.21-3.24
2. Sun SwingSet program modified to present both Classic and Aqua Mac options. (The executable is ModifiedSwingSet.jar; other files need to be in the same folder)
3. Demonstration of creating components with absolute positioning using JBuilder RAD tools
4. GridBagConstraints example - handout and executable
5. ATM System on web
6. Transparency showing grid bag structure of overall ATM

I. Introduction

A. At the outset of our discussion, it is important to again note what we discussed at the start of the course about different kinds of stakeholders in a software project. Recall that a stakeholder is anyone who has a legitimate stake in the outcome of a project.

1. For a typical software project, there are four kinds of stakeholders. Who are they?

   ASK

   a) Users - those who will eventually use the software.

   b) Clients - those who decide to have the software developed, and pay for doing so.

   c) Developers - those who actually produce the software.

   d) Development managers - those who oversee the work of the developers.

2. A common mistake in software projects is to consider the needs of the clients, but not the needs of the users who will use the software. (An even worse mistake is to ignore both in favor of the needs of the developers!)

B. There is a growing awareness in the software industry of the need to very consciously think about the eventual users of the software - i.e. not just the client who is paying for it, but the people who will actually use if (who may be employees or customers of the client.) This leads to a key concept called User-Centered Design.

1. One key notion in UCD is involving users in the design process.

2. Another key notion is bearing in mind user characteristics in the design process. The book noted several characteristics of potential users of a software system that should be borne in mind.

   ASK

   a) Their goals - why are they using the software?

   b) Their demographics - what is their age, education, language and culture, etc?

   c) What is their knowledge of the software domain? (Employees of a client are typically more knowledgeable in this regard than customers - but not necessarily if the software is controlling a complex system.)

   d) What is their knowledge of using computers?

   e) What is their physical ability? Designing software systems to be easily usable by people with physical handicaps is important:

      (1) Visual handicaps, including blindness, need for large fonts, color blindness.

      (2) Hearing handicaps

      (3) Need for alternatives to using a mouse

      etc.

C. Another key idea in UCD is the notion of a use case.

1. The book discusses use cases in this chapter. We considered them earlier, because of their crucial role in driving the software development process.

2. In developing a system incrementally, it is good practice to first focus on the "central" use cases - i.e. the ones that represent the reason why the system exists.

   a) Note that we did this for the course project - renting and returning items are clearly the central cases for a video rental store. (We included the item status report to make initial testing possible)

   b) A related idea is that of developing the "high-risk" use cases first.

D. One of the most important factors in determining the success of many software systems is the quality of its user interface - the way in which human users interact with the program.

1. This may be:

   a) A command-line interface.

   b) A graphical-user interface

   c) A special hardware interface (e.g. for an embedded system, such as our ATM example.)

   d) In some cases, of course, the user interface may seem not to be an issue at all, because the program does not interact with users directly (e.g. network protocol software that interacts only with other software). Even so, there is typically some kind of user interface involved for adjusting parameters, etc, and the functionality of the software eventually impacts the functionality of the software that users do interact with.

2. The quality of the user interface impacts a program's success in at least two ways.

   a) Users prefer (and will purchase) programs that have a user interface that they perceive as best meeting their needs.

   b) In some cases, human safety or even lives may be at stake - poorly-designed UI's have led, in some cases, to people dying. Examples?

ASK

(1) Therac-25

(2) British Midlands flight 92 example (page 233 in text.)

(3) Shooting down of an Iranian passenger plane by the USS Vincennes (page 264 in text)

E. In this lecture, I want to deal briefly with three issues:

1. Fundamental concepts of user interface design

2. The process of designing a user interface

3. Facilities in Java for constructing a user interface, including an overview of the differences between awt and swing, and some discussion of the use of layout managers.

II. Fundamental Concepts of UI Design

A. The topic of UI design is a **huge** one.

1. A focus of graduate-level programs, including PhD in CS programs

2. A broad field, encompassing several fields in addition to CS, including

a) Psychology

b) Library/Information Science

c) Education

d) Communications

e) Technical Writing / English

f) Art

3. A classic work in the field: Shneiderman, Ben. *Designing the User Interface*. (3rd ed, 1998: Addison Wesley)

4. I certainly wouldn't claim much expertise in this area, so I will rely more heavily than usual on your book.

B. One key concept is recognizing that a good UI has two properties which can conflict with one another: Usability and Utility

1. Definitions?

   ASK

   a) Usability has to do with the ease of using the UI

   b) Utility has to do with the functionality of the UI - what can the user do with the software?

   c) Why are these two sometimes in conflict?

      ASK

2. What are some key aspects of usability, as discussed in the book?

   ASK

   a) Learnability - including provision for both novice and expert users

   b) Efficiency of use (not the efficiency of the software - but the amount of work a user must go through to use the software in terms of selecting options, responding to modal dialogues, etc.)

   c) Effectiveness of error prevention / detection / correction

   d) Acceptability - do users like to use the system?

C. In the world of UI design, there are some key terms which you should be familiar with (These are discussed on page 246 of the text).

   ASK class for definition for each

1. Dialogue (as distinct from "dialog box")

   The interaction between the user and the system

2. Control or "widget"

   A visible UI component - menu, button, etc.

3. Affordance

   The set of operations the user can do at a given point in time. UI designers would say "A button affords clicking"

4. State

   At any given point in time, what the user sees and can do (the set of widgets and the affordance at some point in the dialogue)

5. Mode, modal dialog

   A state in which the affordance is restricted to a limited set of options. A modal dialog is a dialog box that requires a user to "satisfy" it before doing anything else. As a general rule, modes and modal dialogs should be minimized, but are sometimes useful.

   a) Example: ASK

      "File save" and "Print" dialogs are often modal - once the user has decided to save or print a file, it makes little sense to allow further changes until the action is complete.

   b) Note that modal dialog boxes often have a "Cancel" to allow the user to get out of the mode without actually doing anything.

6. Feedback

   The response from the system to an action performed by the user

7. Encoding Technique

   The way information is presented to a user - can be audibly, visually, or both

   Note that great care needs to be used in selecting encoding techniques to allow for physical handicaps, to preserve privacy, and to avoid annoying users.

D. Finally, we should note that there are many principles of good UI design.

1. The text gives 12 usability principles in section 7.5, plus an illustration of using them to improve a defective GUI. You should study this section carefully. You will have an opportunity to apply these on a homework problem.

2. Another writer gives a more pointed set, and another illustration.

   TRANSPARENCY - Braude figures 3.21-3.24

III. The Process of Designing a User Interface

A. This will not be an attempt to give a detailed process for designing a UI. Rather, we will note some tools that can be helpful

B. In general, a UI is easier to learn if it follows established conventions - e.g.

   1. A File menu is the standard way of specifying file-related operations, including print and quit. (Even if there is nothing else that makes sense in a file menu, quit is still normally there.)

   2. If it is meaningful to let the user undo an action, that will normally be the first option in an Edit menu.

   3. If it is meaningful to include "cut and paste" in the UI, then an Edit menu with these options is needed.

   4. Normally, the program should display just one window on the screen (to avoid confusing the user). The exception would be a "document-centric" program that lets the user work on multiple documents at once - in which case the program will typically have one window per document, and a Window menu that includes the option of selecting different windows.

   5. Often, menus will have keyboard shortcuts. There are certain traditions related to these - e.g. the shortcut "S" is normally used for Save, "O" for open, "W" for close, "C" for copy, "V" for paste, "X" for cut, and "Z" for undo.

C. A good starting place for the design of a UI is the use case model for the system.

   1. Obviously, the UI must make provision for each use case

   2. If the use cases are simple, it may be desirable to associate either a button or a menu option with each use case.

      Note: Some operations - such as opening, saving, or creating a file - are traditionally done using options in the File menu. Most other operations are better associated with buttons

3. If the use cases involve more complex operations, it makes sense to allocate a screen (or series of screens) to each. In this case, a button is typically used to initiate an operation.

4. Often, it is meaningful to group use cases into groups of closely-related operations, which then might have a common starting point (e.g. individual panes within a tabbed pane.)

D. Often, a statechart can be used as a design tool.

1. Each state would correspond to a single screen - or perhaps a state of a screen (e.g. with certain operations enabled and others not.)

2. Transitions between states correspond to user actions such as clicking a button.

   The UI should be designed so that state transitions that would lead to problems are not possible.

3. You have already done some work representing a GUI using a statechart on your project.

IV. Java awt and swing

A. As you know, the standard Java libraries include two toolkits for building user interfaces: java.awt, and javax.swing. Although swing builds upon awt, and uses some parts of it, the two facilities are distinct enough that it is best to use one or the other in any given program, rather than trying to mix them indiscriminately.

B. AWT

1. awt was the original user-interface facility in Java.

2. It is based on what are called heavyweight components.

   a) Each of the basic awt Components (buttons, labels, etc.) has an analogue in the GUI facilities of the various platforms on which java runs. Indeed, the awt was designed around a common subset of components that would be expected to be available on any platform that supports java - a sort of least-common denominator.

   b) In terms of implementation, each awt Component has a native *peer* component, which is furnished by the specific platform, and is part of its GUI toolkit. A *delegation* model is used, whereby

(1) Whenever a java.awt Component is created, a corresponding peer component is created as well - e.g. if a java.awt.Button is created on an OS X Macintosh, then an Aqua button is created as well; if it is created on a Linux platform, then a Motif button is created as well, etc.

(2) When the java program performs some operation on a Component (e.g. paints it or sets its label or color), the awt Component simply asks its peer to perform that operation. Thus, for example, if a button needs to be painted at a certain place in a window, the java.awt.Button tells its native peer to paint itself at that place; if a program sets the label or color of a button, then the java.awt.Button tells its native peer to set its label or color accordingly, etc.

(3) An important consequence of this is that awt components necessarily have the same "look and feel" as the platform they are running on - since they are implemented by platform-specific peers. This helps make Java applications look like they "belong".

(4) Another consequence of this is that the operations supported by an awt component must have a native counterpart in the peer - which leads to some restrictions on the functionality available with awt components.

Example: The standard awt Button can display a textual label, but not a picture. This is because, on some platforms, standard buttons can display either text or pictures, while on others they can only display text. The awt Button provides only the functionality that can reasonably be expected to be available on *any* platform to which java is ported.

This is not followed rigidly. For example, native buttons on the Macintosh Classic OS platform (OS 9 and before) always have a gray background. For this reason, if one tries to set the background color of a Button on a Macintosh Classic platform, the operation is simply ignored because the peer cannot do it.

C. Swing

1. In contrast, swing is based on <u>lightweight components.</u> Most swing components are written entirely in java, and don't have a native peer. Instead they do everything for themselves - e.g. when a swing JButton

is painted, the code in class JButton actually draws the button as a series of lower-level graphic operations such as drawing a rounded rectangle, etc.

a) A consequence of this is that the functionality of a swing component does not depend on what's available on the underlying platform (beyond the provision of certain fundamental low-level capabilities that any platform furnishes.)

b) Another consequence of this is that swing components can have any "look and feel" - not necessarily the "look and feel" of the OS platform they are displayed on.

c) It is at this point that the designers of swing made a critical design choice. Rather than designing swing around a single "java look and feel", swing components are designed around the notion of a "pluggable look and feel". Each pluggable look and feel (plaf) is basically a set of specifications for what a given type of component should look like - e.g. what should a button look like? what should a text field look like? etc.

(1) The java distribution for most platforms includes several different pluggable looks and feels - typically including "Metal" (the "java look and feel", "Motif"(the look and feel typically found on unix platforms, including Sun's Solaris) and the native look and feel of a specific platform - e.g. "Mac" in the version of swing written for macs, "Windows" in the version of swing written for wintel machines, etc.

(2) When a program running swing starts up, code within the program selects the look and feel to be used.

(a) You've seen code for this in the main method generated by JBuilder when you create a swing application.

(b) The default action is to use the look and feel of the platform on which the program is running. This is normally a good choice because:

i) The user is used to this

ii) The program looks like it "belongs" on the system it is running on.

(c) Normally, the look and feel is not changed while a program is running, of course.

(3) Although Java distributions come with a few standard looks and feels (along with the look and feel of the underlying platform), there is no reason - in principle - why a programmer could not create his/her own. The major challenge would be that it's a big job - and probably not worth the effort given that using a standard look and feel is less jarring to users.

2. For demonstration purposes, Sun has written a program called "Swing Set" which demonstrates the different looks and feels available with swing. I modified it to allow demonstrating both Macintosh looks and feels: the old "Classic" look and feel (OS 9 and before) and the new "Aqua" look and feel (OS X).

   DEMONSTRATION

   Change between looks and feels for Buttons, RadioButtons, Checkboxes, ComboBox, ProgressBar, TreeView - also note changes to tabbed pane.

   Note: One normally wouldn't change looks and feels while a program is running like this - this is only for illustration.

D. When designing a user interface, one choice you need to consider is whether you will use awt or swing.

1. A major advantage of swing is that it is a lot richer, since it includes many more kinds of components, and the components themselves often have a richer set of behaviors.

   (Note the list of components you made in lab)

2. A disadvantage of swing is that a number of things are done in a more complex way (in order to support the richer set of options), so it's a bit harder to use than awt.

   A simple example: an awt Frame is a container to which you can add visible components directly; a swing JFrame has a content pane to which components are added, so you need to explicitly call getContentPane() on the frame to get the pane to which you add components.

3. We have been using swing, both in CS112 and in CS211. However, there are some older browsers whose Java installation only supports awt - so there may be reason, on occassion, to use awt.

4. As noted previously, swing is actually built on top of awt.

   a) All swing components inherit from java.awt.Container

   b) A JFrame inherits from java.awt.Frame and thus has a peer, and a JApplet inherits from java.awt.Applet and thus has a peer - these peers form the "hook" that connects a swing-based GUI to the native GUI of the underlying platform. (The top-level frame or applet container is native, but swing is responsible for everything that appears inside of it.)

   c) Swing uses awt layout managers and events - it doesn't have its own kinds of layout managers or events.

      Nonetheless, it is usually a bad idea to mix the two kinds of visible components (regular and menu) in one program.

V. Layouts and Layout Managers

A. Another important issue in the design of a user interface is the visual layout of the components, which has both aesthetic and functional importance.

B. Java supports two different ways of laying out a GUI:

   1. The use of absolute positioning

   2. The use of layout managers

   3. In general, it is not desirable to mix these in one program - too many things can go very badly wrong when one does this.

   4. Absolute positioning is, in some respects, easier to use with a RAD tool - you just need to drag a visible component to be the size you want it to be.

      DEMO: Create a GUI using RAD tools

      Nonetheless, absolute positioning is not the preferred approach for two reasons:

      a) Platform-independence issues: an absolute design that is right for one platform may end up looking weird (e.g. cutting off portions of components) on another.

b) In general, it is hard to avoid a "ragged" look when using absolute positioning.

c) We will therefore devote the rest of our time here to discussing how to use layout managers

C. The standard java library supplies a number of layout managers.

1. The package java awt supplies five, which can be used with either awt or swing.

   a) BorderLayout

   b) CardLayout

   c) FlowLayout

   d) GridLayout

   e) GridBagLayout

2. Swing supplies some additional ones - but these are used internally by swing components, and are not generally usable.

3. JBuilder supplies several additional layout managers which can be used in projects produced with JBuilder (but which are not standard java)

   a) BoxLayout2 (a "wrapper" around the swing BoxLayout)

   b) OverlayLayout2 (a "wrapper" around the swing OverlayLayout)

   c) PaneLayout

   d) VerticalFlowLayout

   e) XYLayout

4. It is possible to write your own layout manager - although this is a non-trivial task.

D. It is helpful to understand exactly how a layout manager works

1. Every visible component (button, label, etc.) has several different size values accessible by methods of the component. Each size value includes a width and a height.

a) The sizes include:

(1) A minimum size - the smallest amount of space that can be given to this component and have it still be functional (e.g. if its a label, showing all of its text.)

(2) A preferred size - generally the same as the minimum size for individual components.

(3) A maximum size. (The component can be given more space, but won't put anything in it. For many components, the maximum size is infinite.)

(4) A current, actual size (accessible via getSize(), changed via setSize() or setBounds())

b) The minimum, preferred, and maximum sizes are just hints to the layout manager - they are not enforced in any way. (A layout manager can make a component smaller than its minimum size - in which case some of its content will be cut off. Making a component bigger than its preferred size results in a "stretched" appearance.

c) The minimum, preferred, and maximum sizes may depend on various other properties of the component - e.g. if it displays text (a label, field, or button) then its current text and font. Thus, changing the font of a component that displays text will almost certainly change both its minimum and preferred height and width, while changing its textual content will change its minimum and preferred width.

d) If a component is a heavyweight component, the task of calculating minimum, preferred, and maximum sizes is delegated to its peer. Thus, valid values are not available until after the peer is created (which occurs during the layout process.)

2. The sizes for a container are calculated by the layout manager from the sizes for the components it contains. This involves getting the appropriate size value (e.g. minimum, preferred) for each component in the container, and then figuring out what the overall size for the container needs to be to at least honor these values.

a) Example: Minimum size of a BorderLayout:

Width is the maximum of the minimum width of the North component, minimum width of the South, or sum of the minimum widths of the West, Center, and East components plus two times the horizontal gap.

Height is the minimum height of the North component plus the minimum height of the South component plus the maximum of the minimum heights of the West, Center, and East components - in any case plus two times the vertical gap.

b) Example: Minimum size for a GridLayout

Since all the cells in the grid are the same size, each component is asked for its minimum size, and then the width and height of a cell are established as the maximum of all width and height values.

The overall width and height are determined by multiplying the width and height of an individual cell by the number of columns / rows, and then adding in an allowance for horizontal and vertical gaps.

3. The layout process is normally initiated by calling pack() on the top-level frame (though there are other ways).

a) First, the overall size needed for the frame is calculated using the preferred size as calculated by the layout manager - which in turn asks the layout manager for any nested containers to calculate a preferred size for the container it manages.

b) Then, the layout manager is asked to apportion any excess space.

(1) BorderLayout:

If the North or South component needs less width than the width actually allocated to the container, it is "stretched" to fill the allocated space. If the sum of the West, Center, and East component widths is less than the width actually allocated to the container, then the Center component is "stretched" to make the total add up.

The height for the middle three components (West, Center, East) is calculated by subtracting the height needed by the North and South components. Then each of the middle components is "stretched" (if need be) to fill this height.

(2) CardLayout

All components are "stretched" to the match the size needed by the biggest card.

(3) FlowLayout

All extra space is added at the end of the layout - each component gets just the space it needs, and no more

(4) GridLayout

The actual cell size is determined from the total size of the container divided by the number of columns / rows - after allowing for horizontal and vertical gaps.

All the cells must have the same size. Thus, any component needing less than the calculated cell size is "stretched" to fill the calculated cell size.

c) How the component deals with extra space depends on the component - e.g.

(1) If the component is itself a container, its layout manager is asked to apportion the extra space.

(2) Buttons are always stretched out of shape.

(3) For a label, the alignment property determines whether the text is bunched at the left of the assigned space (with the right being empty), or at the right, or is centered.

(4) Etc.

d) Note that the layout process described here can never result in a component getting less space than it needs, but often results in a component getting more space than it needs, and thus being "stretched". The exception is FlowLayout, which simply puts the extra space at the end.

Thus, it is common practice to "wrap" a component that doesn't look good when "stretched" in a FlowLayout to protect it from being stretched. (When the FlowLayout is stretched, the extra space goes at the end of the layout, rather than the component being stretched).

e) If the user resizes a container after it is laid out, the same process is used to calculate the new sizes of the various components. Note that this can result in a component getting less space than it needs, if the user shrinks the size below the minimum size calculated when the container was packed.

f) Note that, if the size needed by a component changes after the container is laid out, it may be cut off.

(1) Example: changing the content of a label after its container is laid out could result in some text being lost if the new content is bigger than the old.

(2) Example: if a component is flagged as invisible when the layout is done (setVisible(false)), most layout managers won't assign it any space. If it is subsequently made visible, it may well be cut off.

E. You'll notice that, in the discussion thus far, we have said little about the GridBagLayout - which is the most sophisticated layout manager, but also the most cumbersome to use.

1. Like a GridLayout, a GridBagLayout layout manager treats its container as a grid of cells. However, rather than making all columns have the same width, each column can have a different width. Likewise, each row can have a different height. (It remains the case, of course, that all all cells in the same column have the same width, and all cells in the same row have the same height.)

a) During initial size calculation, a GridBagLayout manager assigns a width to each column based on the widest width needed by any component in that column, and a height to each row based on the tallest height needed by any column in that row.

b) Each row and column can have a weight associated with it, which determines how extra space is allocated - e.g. if all columns have the default weight of 1, then extra width is allocated equally among all columns; but if one column has a weight of 1 and all the rest have weights of 0, then all the extra width is given to the column whose weight is 1.

2. An individual component can occupy more than one cell - i.e. it could be (say) 2 cells wide and 3 cells high.

3. The positioning of each component within the grid bag is specified by a GridBagConstraints object that is specified when it is added to the grid bag.

4. EXAMPLE program using GridBagLayout / Constraints:

   HANDOUT

   RUN program - highlight the "Observe" points on pp 3-4

5. Individual fields of the constraints object

   a) gridx, gridy - column and row for upper-left corner of component (origin (0, 0)). The total size of the grid bag is determined by the maximum row and column specified for any component

   b) width, height - the number of columns and rows spanned by the component (Normally 1, 1).

   c) weightx, weighty -used to set the weights for column and row - the weight of any column is the maximum weight of any width 1 component in that column and likewise for rows. (If a component spans multiple rows or columns, its weight specification is distributed among the columns/rows)

   d) anchor, fill - controls how a component is positioned within a cell if the cell is bigger than the space the component needs:

      Possible fill values:

      NONE - The component is never stretched
      HORIZONTAL - If the cell is wider than what the component needs, it is stretched horizontally; but never vertically
      VERTICAL - If the cell is higher than what the component needs, it is stretched vertically; but never horizontally
      BOTH - The component may be stretched either horizontally or vertically if its cell is bigger than the space it needs

      Possible anchor values: (relevant if the cell is bigger than what the component needs, but fill specifies no stretching)

      NORTHWEST, NORTH, NORTHEAST, WEST, CENTER, EAST, SOUTHWEST, SOUTH, SOUTHEAST

   e) insets - allows associating an Insets object with the component, which specifies space to be left between the component and the edge of its cell

f) ipadx, ipady - allows "padding" the component beyond its normally preferred size - the component is stretched to include this extra space

6. The ATM Example uses a GridBagLayout for the simulated ATM

SHOW ON WEB - DEMONSTRATE INSERTING CARD, WITHDRAWING AND DEPOSITING MONEY TO SHOW THE VARIOUS COMPONENTS

a) Can you see the grid structure here?

ASK

b) Show transparency of structure

c) Show code for ATMPanel in the simulation package - note symbolic constants used to set up the layout. (I spent a bit of time "tuning" these.

d) Any other layout managers in evidence as you look at the overall ATM?

ASK

(1) The keyboard uses a GridLayout with 3 columns and 5 rows

(2) The display uses a GridLayout with 1 column and 9 rows - each line of text on the screen is a label associated with one or the rows in the grid

(3) The operator panel uses a FlowLayout