14

Designing the Graphical User Interface

CERTIFICATION OBJECTIVE

- Creating a Usable and Extensible GUI

CERTIFICATION OBJECTIVE

Creating a Usable and Extensible GUI

There are several key aspects of GUI design that you need to consider when designing and implementing the GUI for your project. At a high level, they can be broken down into two main areas of focus:

1. Designing the GUI to be usable and friendly from the end user's perspective.
2. Designing and implementing the GUI to be reliable, and maintainable from the programmer's perspective.

This chapter will focus almost entirely on the *first* point—ease of use for the end user. We start with a very brief overview of the technical issues you probably want to address in implementing your GUI for this project. After that brief overview, we dive into the topic of usability.

An Overview of Technical Considerations for Your GUI

Most of your GUI work on the exam assignment will be focused on usability. But for the final review, you might be asked to justify not just the user-friendliness, but also the technical considerations you took into account when designing and building your GUI. This section gives you a brief overview of some of the technical issues you need to keep in mind.

Required Technologies

Your instruction packet will probably require you to use certain technologies to implement your GUI for this project. If, for instance, your instructions indicate that you are to use Java Swing components and specifically the JTable component, not only do you have to use them, but you also need to use them appropriately. Before jumping in to implementing your GUI, you need to understand the strengths and weaknesses of the technologies you are using. In addition, each of the required technologies is meant to be used in a certain fashion—for instance, if you're going to use a JTable, you'll want to use the appropriate models and listeners. The bottom line is, don't use a widget until you really understand how Sun intended for you to use it.

Model–View–Controller

Your exam instructions will probably say that the GUI you build should be flexible and relatively easy to extend. If so, you'll probably end up considering the Model–View–Controller (MVC) design pattern. We recommend that you *do* consider the MVC approach. If you are familiar with it, so much the better. If you are not, this is a good opportunity to study it. The MVC pattern has plenty of benefits:

- It's very popular, and you're bound to run into it sooner or later.
- It anticipates that end users will ask for iteration after iteration of changes to the GUI design, and it reduces the development impact of those iterations. (You *know* how those end users are!)
- It scales well to large teams.
- It anticipates Java's "write once run anywhere" philosophy, reducing the effort required to port your GUI layer to additional environments such as browsers or mobile devices.

Event Handling, Listeners, and Inner Classes

If you're instructed to use Swing (and we can virtually guarantee you will be), you must understand the Listener paradigm. Be certain that you understand how Swing components are meant to handle events, and how components relate to their models and their listeners. In addition, you should understand how inner classes are used to help implement these capabilities.

Introduction to Usability Design

Traditionally, the assessors for the developer's exam have given a good deal of weight to the quality of the GUI. To pass the exam, your GUI should embody a host of wonderful attributes, including

- It should follow presentation standards.
- It should be easy to learn and easy to use.
- It should behave as GUIs are expected to behave.

The rest of this chapter covers key aspects of usability design and implementation for the GUI portion of your project. As an added bonus, this chapter discusses GUI design aspects that will be applicable across most of the GUI projects you encounter. Once again, we are approaching this topic with our infamous 80/20 perspective; this chapter provides the core information you need to design GUIs that are easy to learn, easy to use, and ergonomically friendly. There are eight steps to designing a great GUI:

1. Understand the business function and build use-cases.
2. Follow the general principals of good screen design.
3. Choose the appropriate widgets for your GUI.
4. Create the basic page layout(s).
5. Create appropriate menus and navigational elements.
6. Create crisp and informative GUI messages.
7. Use colors responsibly.
8. Test your GUI iteratively.

1. Use-Cases and the Business Function

The Sun developer's exam is by its nature an artificial exercise. We all understand that there are no real end users and no real business with real issues being addressed here. The rest of this section is written assuming that you're creating a solution for a real scenario. So, for the exam, you'll just have to pretend that *you* are the user, the business manager, etc. Even though you're a one-person band, you can follow this process—at least for the exam.

Interviews, Observation, and Iteration

A GUI will *always* be better if it's designed with the help of the end-user community. No matter how many businesses you've helped to automate, or how many killer GUIs you've built in the past, end-user input is essential. Although there are many ways of interacting with the end user, the three ways that offer the best return are

- Observing the end user performing *the process that you hope to automate.* From now on we'll just call it *the process.*

- Interviewing the end user about the process that he or she performs—what information is used, what decisions are made, what steps are taken, and what is created.
- Reviewing your results, and refining your implementation, with the user, over and over again at every stage of development.

Creating Use-Cases

A very effective approach to designing a GUI is to create "use-cases" with the user as you work through the observation and interview stages. Use-cases let you encapsulate the transactions that the end user performs over and over again in his or her job.

Let's say that you're creating a system to help customer service representatives (CSRs) at a company that sells PCs over the phone. After talking with the CSRs and watching them work, you might discover that they perform the following tasks over and over again in the context of working with their clients:

- Create a new customer record.
- Modify information in an existing customer record.
- Place a new order.
- Modify an existing order.
- Cancel an order.

Each of these activities can be considered a *use-case*. Once you've generated a list of use-cases for your application, the next step is to flesh them out. We like to use 4 × 6 cards for this. Each use-case is listed on its own card, and then for each card we add the following headings:

- **Find** How do you find the information you need to perform the use-case?
- **Display** What information is needed for the use-case to be completed?
- **Verification** What processes support verifying that the use-case is completed properly?
- **Finalization** What processes and information are necessary to complete the use-case?

The next step is to work with the end users to answer the four questions listed on each use-case card. When the cards have been completed and reviewed, they form the basis for designing the GUI.

Screen Mockups

The next step is to use your deck of 4 × 6 cards to generate *hand-drawn* screen mockups. Don't worry about making these mockups look good—that's handled later. Just get them down on paper quickly; they're just temporary. It's tempting to get ahead of yourself here and want to jump in and start writing code. Avoid the temptation! If done correctly, this first whack at screen design will produce screens that will absolutely *not* be what you'll want the final system to look like. In this phase, you want to quickly sketch out a rough screen for *every* heading on *every* card. If we've done the math right, that means you'll have four mock screens for every use-case; Find, Display, Verification, and Finalization.

It's hard not to get ahead of yourself here, because you'll quickly realize that a lot of these mockup screens look a whole lot like each other. That's a good thing. By reviewing these mockups with the end users, you'll discover that with just a little tweaking you can solve many different use-case steps with a single display. In our previous example, we had five use-cases, so it might seem reasonable to expect that you can represent all 20 different use-case steps with three or four displays.

2. Principles of Good Screen Design

Once we've got a rough idea what the system's individual displays ought to look like, it's time to move to the next level of design. At this stage in the design, our goal is to create mockup displays that do more than simply satisfy the requirements of the use-cases. We also want to design screens that will be easy to learn, easy to use, and will not irritate the end users over time. Here is a list of principles that will assist you in creating screens that your users (*and assessors*) will love.

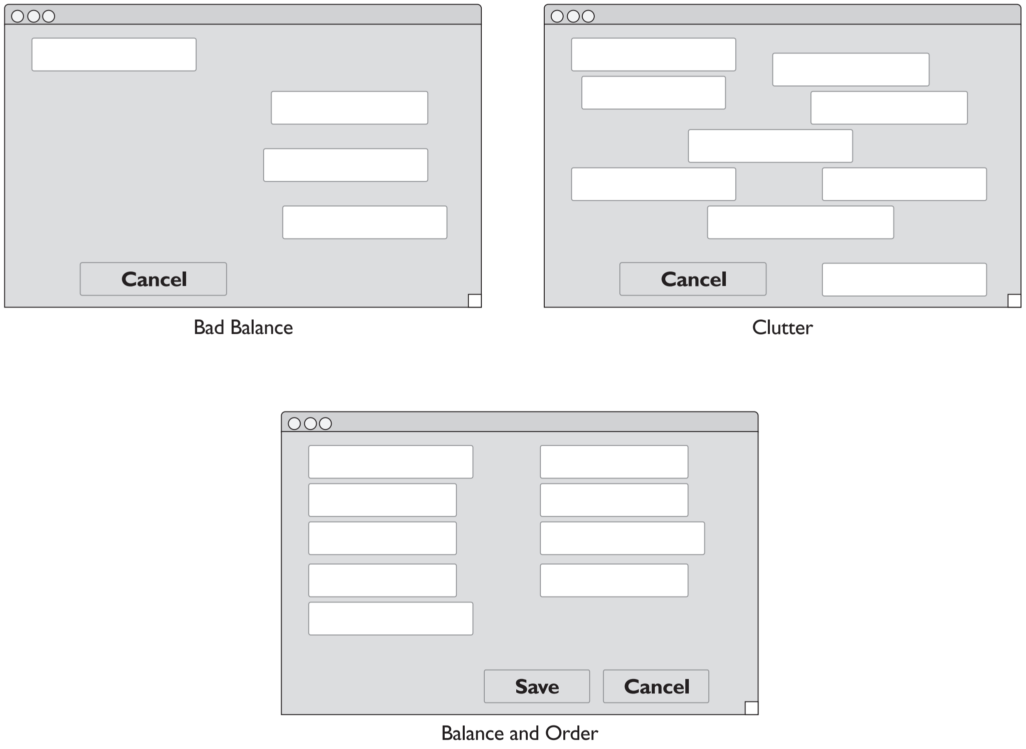
Balance and Clutter

Well-designed displays tend to be balanced. By balanced, we mean that the content is approximately balanced left to right and top to bottom. Another attribute of good-looking displays is that they avoid the feeling of being cluttered. We return to the issue of clutter again later, but for now we mean that the screen elements should be neatly aligned. Figure 14-1 shows some examples of cluttered and poorly balanced displays, and then an orderly and well-balanced display.

Logical Groups and Workflow

Often a display can be thought of as many individual elements that can be placed into a few logical groups. For instance customer name, street address, city, state, and ZIP code are all individual elements, but it's natural for users to view these individual elements as a group, thought of as "customer address." Grouping elements together in a natural way is a good practice—there will be less mental strain on the user, data entry errors will be reduced if the display's tab sequence produces the shortest possible "travel" between elements, well-ordered groups tend to be more visually appealing, and, finally, natural groups are easier to learn.

FIGURE 14-1 The dos and don'ts of an orderly and balanced display



You should also consider conditioned scanning patterns. In Western cultures, information is typically presented from left to right, and from the top down. These sequences are not universally recognized, however, so you should consider local cultural factors when designing a display.

Navigation and Commands

Good GUI displays typically let the user issue a variety of commands to the system. For now, we split GUI commands into two broad categories: commands that cause an action to take place within the current display (*action commands)*, and commands that make the system jump to a new display (*navigational commands*). A good rule of thumb is that action commands can occur wherever related elements are being displayed, but that navigational commands will appear only in the menu bar or toolbar, or at the bottom of the display.

When designing screen commands, simple language is the best. As a developer you know, for instance, that displaying the contents of a customer record on a display may actually require several programming steps. You don't want a button that says: "Create a search string, query the database, verify that good data was received, and, finally, display the result." Instead, you probably want something like a command button that says: "Find Customer." We'll talk more about good messages in a later section.

3. Choosing Your Widgets, JTable, and What Army?

We already mentioned that you'll probably have to use standard Swing components to implement your GUI, and that specifically you'll have to use the JTable component for a key part your main display. In addition, the second part of the exam (the follow-up essay) may ask you to describe why you made the widget (component) selections you made. Swing is a very rich GUI toolkit, and the instructions leave you with a lot of leeway in deciding which Swing components you should use for most of your application. In this section we describe many of the more common Swing components that are available, and when you should consider using them.

- **JLabel** Labels are strings of text used to identify other widgets. They are typically placed to the left or above the widget being labeled.
- **JButton** Buttons are rectangular-shaped widgets that are used to initiate actions by the system. A button can be used to initiate either an action

command or a navigational command. The nature of the command controlled by the button is typically displayed as a "label" inside the boundary of the button.

- **JTextField** and **JTextArea** Text fields and text areas are rectangular-shaped widgets that are used to either display textual data supplied by the system or as an area into which the user can type in data. Use a text field when you need no more than a single line, and a text area when the amount of data might exceed a single line. Text areas can have scroll bars (via the support widget called JScroll Pane), while text fields cannot. Text fields and text areas are typically festooned with a label placed above or to the left of the widget.
- **JRadioButton** This widget is named after the buttons found in car radios back in the good ol' days. These mechanical arrays of buttons were designed so that each one, when pressed, would tune the radio to a particular station. One of the key features of these radio buttons was that only *one* button could be depressed (in other words, *be active*) at a time. When an *inactive* button was pressed, the currently *active* button would be undepressed, and functionally move to the inactive state. The radio button widget works in much the same way; use it when you want the user to *choose one and only* one option from a list of options.
- **JCheckBox** This widget is often associated with radio buttons. It has a slightly different look and feel, and a different (albeit related) functionality. Like a radio button, the check box widget presents the user with a list of options. Unlike the radio button widget, the check box widget allows the user to select as many or as few choices as she or he wants. So, radio buttons are mutually exclusive, but check boxes are not.
- **JList** This widget presents the user with a list of entries, set inside a rectangle that can have a scroll bar. The entries are arranged in rows, one entry per row, so that the user can use a vertical scroll bar to search the list if there are more entries than can be displayed at one time. The list widget allows the user to select as few or as many entries as she or he wants.
- **JComboBox** This widget is part text field, part list (a *combo*, get it?). When not selected, the combo box resembles a text field with a down arrow button appended to its right end. The user can choose to key data into the field portion of the widget (like a text field), or choose the down arrow, which will cause a temporary list-like widget to appear. The user can select an option from this

temporary list in the same fashion that a normal list selection is made. Once the selection is chosen, it is placed into the text area of the widget and the list portion disappears.

- **JSlider** These widgets present a (typically horizontal) control that lets the user adjust the setting of any system feature whose possible values can be mapped to a range. Typical uses for sliders would be to control display brightness, or for volume (sound) control.
- **JProgressBar** These widgets present a (typically horizontal) display that allows the system to interactively indicate the progress of some system function that takes a noticeable time to complete. This widget is used to give feedback to the user so that she or he will know the system is still working, and roughly how much longer before the system will be finished with its task.
- **JTabbedPane** This widget allows the developer to pack a lot of display functionality into a small space. The analogy is that of looking at the tabs at the top of an open file drawer. When a tab is selected, an entire window of display elements associated with that tab is displayed to the user. When another tab is selected, the current tab display disappears and is replaced with a new set of elements. This widget is typically used when you need to support many infrequently used display elements. Application preferences or parameters are typically accessed via a tabbed pane widget.
- **JTree** This complex widget allows the system developer to create a traversable tree structure similar to what is presented by the Macintosh Finder or the Windows Explorer applications. This widget allows for arbitrarily large data structures to be represented and accessed. Trees are often used to represent directory structures on a hard drive or for a computer network, or any other data structure that involves nested lists of information.
- **JTable** This very complex widget is used to display and update arbitrarily large tables of information. In this usage, a table is typically a two-dimensional array of rows and columns. Generally, a table is structured so that each row of elements represents a collection of related information (often a row from a database). In this scheme, each column represents an element in the collection. You'll probably be required to use JTable in your project.
- **JMenuBar** Almost all good GUI displays include a menu bar widget. This (usually horizontal) widget is most commonly found at the top of the screen

directly under the title bar. The menu bar lets the developer arrange a wide variety of commands and system settings into logical groups. Menu bars are a part of almost every GUI display, and we look at them more closely in a few sections.

- **JToolBar** The toolbar widget is typically located directly beneath the menu bar. It displays a series of icons, each of which acts like a button, initiating an action or navigational instruction for the system.

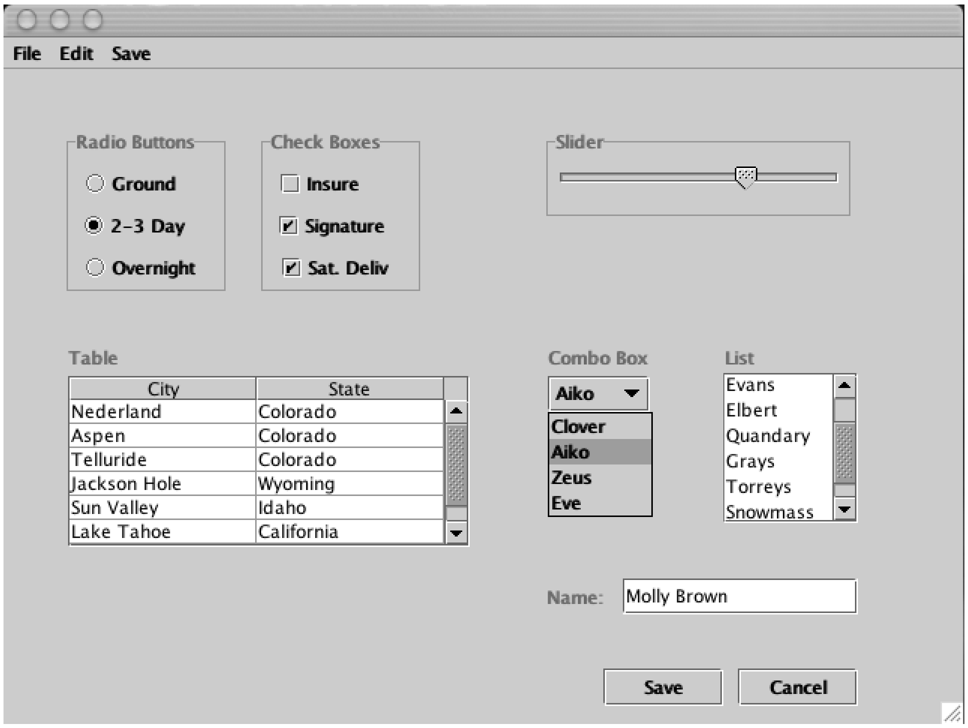
Figure 14-2 illustrates the look and feel of this wonderful array of GUI widgets.

4. Screen Layout for Your Project

Now that we've developed our use-cases, mocked up some trial screens, equipped ourselves with an arsenal of Swing widgets (or components to be proper), and

FIGURE 14-2

Explosion at the widget factory



Radio Buttons: ○ Ground [✓ 2–3 Day] ○ Overnight

Check Boxes: ☐ Insure ☑ Signature ☑ Sat. Deliv

Table

City	State
Nederland	Colorado
Aspen	Colorado
Telluride	Colorado
Jackson Hole	Wyoming
Sun Valley	Idaho
Lake Tahoe	California

Combo Box: Aiko

Clover
Aiko
Zeus
Eve

List

Evans
Elbert
Quandary
Grays
Torreys
Snowmass

Name: Molly Brown

learned a little something about layout principals, it's time to put all of these pieces together! Hooray! Wait, wait, it's still not quite time to warm up your compiler—we're going to do a little more work with paper and pencil first. This phase of the design is concerned with designing the main portion of your GUI displays. The idea is to take the rough displays you designed in phase 1 and apply the rules of phase 2 and the widgets of phase 3 to these displays. When you're working on this phase, the following tips will help you create solid screen layouts:

- Remember, the user's eye will flow from left to right and from top to bottom. As much as possible, the standard path through the display should follow this natural flow.
- Try to make the display as visually pleasing as possible:
  - Don't jam too many elements into a single screen. White space and borders help keep a display looking clean, orderly, and less overwhelming.
  - Group related elements. You'll often want to place a labeled border around such a group.
  - Imagine invisible gridlines running vertically and horizontally through the display and align your groups and elements along these gridlines whenever possible.
- While other arrangements are acceptable, it's almost never wrong to right-justify text field labels and left-justify their respective text fields around the same vertical line. (See Figure 14-3.)
- Place your menu bar and toolbar (if applicable) at the top of the screen (more in the next section).

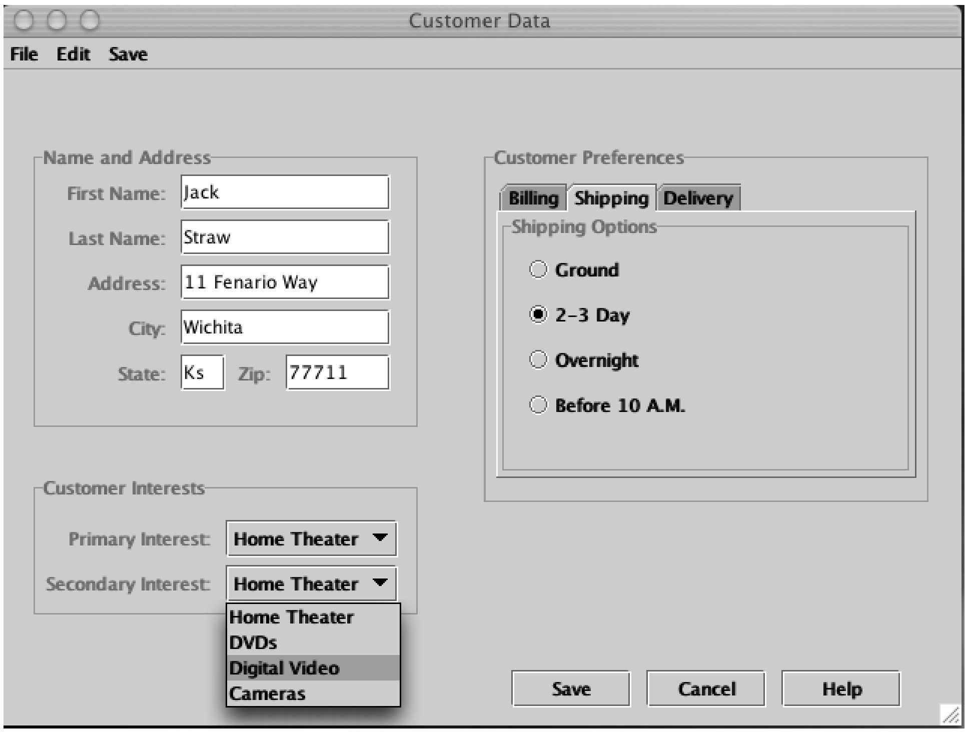
Figure 14-3 illustrates examples of many of the concepts we've been discussing. Notice that the name and address elements are grouped logically, and that they are aligned along a vertical line. The client preferences are accessible through a tabbed pane; this example shows a typical use for a set of radio buttons. On the lower left we've aligned two related combo boxes, and the navigational buttons are horizontally aligned in the bottom right of the display.

5. Menus and Navigation

Just about any standard application has menus and navigation buttons to let the user make choices and move to other windows. You'll need to pay particular attention to

FIGURE 14-3

An example of design elements



your menus and navigation features; no matter how attractive and easy-to-use you believe your GUI to be, you'll still have points deducted if you don't follow standard conventions.

Menus and Menu Bars

Menus are a powerful (and necessary) part of almost all GUIs. We focus our attention on the most common implementation of menus, the *menu bar*. An application's main menu bar is almost always located at the top of the display—sometimes directly under an application's title bar, and sometimes separate from the application's main window and docked to the top of the display.

You're familiar with the standard menu bar. Several of its more consistent entries are typically located toward the left end of the bar and include File, Edit, and View. Each entry in the menu bar represents a collection of related capabilities. Clicking on one of the entries on the menu bar will cause a specially formatted widget (a *menu*) that resembles a list widget to appear beneath the menu bar entry. The entries in these lists each represent a system capability. The most common capabilities available through menu entries are

- A navigational command such as Close (close the current document), Print (move to the Print display to initiate a print session), or Exit (end the application)
- An action command such as Spell Check (invoke the built in spell checker) or Copy (copy the currently highlighted data to the clipboard)
- Alter a system setting or parameter, such as Show Toolbar (displays the application's toolbar by default) or View Normal (display the current data in the default mode).

Within a menu, entries should be grouped in logical subsets, and each subset is typically delineated with a horizontal line or a double space. Menu commands should be left-justified, and it is common and appropriate to display keyboard shortcut commands, whenever they are available, to the right of the menu entry. Each application will have its own unique set of menus on the menu bar, but several of the menus will be very consistent from application to application. These most consistent menus are the File and the Edit. File will vary a bit from application to application, but it will almost always include commands (menu items), for New, Open, Save, Print, Close, and Exit. These commands refer to the current document or project as a whole. Edit can vary also, but will typically include commands for Undo, Redo, Cut, Copy, Paste, Clear, Select, and Find. These *editing* commands are used to modify portions of the active document or project. Not to give anything away here, but not having a standard menu bar with standard menus and menu items will cost you *big time* on your exam score.

Navigational Buttons

The second most common way to provide navigational capabilities within a GUI is through the use of navigational buttons. Navigational buttons are typically placed on the bottom (or sometimes the right side) of the active window. Navigational buttons typically act on the entire active window; examples include the Save button on a File dialog window, or the Print button on a Print dialog window. In both cases, activating the button causes a system action to take place, *followed by a navigation* to a different window in the application. Sometimes a navigational button will serve a solely navigational function such as Close, which closes the current window and returns the user back to the previous window.

6. Messages, Feedback, and Dialog Boxes

Messages and feedback are essential ways of communicating with your user, and your exam assessor will pay close attention to the way you handle keeping the user informed. The clarity, conciseness, and attitude of your messages can have a huge effect on whether users perceive your application as friendly and easy to use.

Messages and Feedback

Messages and other feedback are the primary ways that you (as the developer) have to respond to the user as she or he is using your application. Use messages to provide warnings when something has gone wrong or might be about to go wrong. Use messages to offer more information about activities that the system is performing (such as "37 occurrences found"), and use messages to display the status of an operation ("Search complete, no matches found").

Feedback tends to supply the user with information that the system has generated; for instance, if you key in a customer number and initiate a search, when the system returns with the customer's name and address that information is considered *feedback*. You also use feedback to tell the user that a lengthy operation is in progress, or that there's a problem with a current activity. Feedback can also be as subtle and useful as the blinking cursor bar that lets the user know where he or she currently is on the display.

Here are some tips for messages and feedback that will make your users smile:

- Try to use short, positive words and phrases.
- Use active voice whenever possible: "Print the file by choosing Print now", as opposed to "The file will be printed by choosing Print".
- Minimize the use of jargon or abbreviations in your messages.
- Ranges should be listed from smaller to larger and from now into the future.
- Action verbs should come first: "*Display* active customers" instead of "Active customer *display*".
- If the user selects an option that you think will require more than a few seconds to complete, give the user some sort of indication of status and progress. A system that locks up and gives the user no idea of whether progress is being made or an error has occurred is considered rude and unfriendly.

- If the user makes a mistake that can be caught immediately, it is appropriate to give them some sort of instant feedback such as warning noise or a blinking element.

Dialog Boxes

One of the nice usability features of a GUI is that the user has a great deal of flexibility in terms of sequencing tasks. For the most part, all elements on the screen are available to him or her at all times. However, when you create your *use-cases* and your mockup screens, you might run into situations that call for a strict sequence of events to take place. Counter to the normal GUI flexibility, there will be times when, for a particular action to take place, you have to follow a fixed path. An obvious case is when the user chooses to save his or her work. When the *save* request is made, no other work on the active project should take place until the save is either completed or abandoned. We use the term *nonmodal* to describe a GUI's typical openness. With a fixed-path situation like the *save* operation described above, the term is, not surprisingly, *modal*. In the GUI world, a modal sequence is one that can't be interrupted.

We know you're familiar with the typical sequence of events when you go to save some work, say a text document or perhaps a spreadsheet. When you make the *save* request, the system typically displays a small window in the center of the screen, known as a *dialog box*. Once the *save file* dialog box has been displayed, no other application actions can take place until the dialog box has been dismissed. This locking out of other actions is called *modal* behavior. When you create a dialog box, you have the choice to make it either modal or nonmodal. Always make it nonmodal as the default. However, there are times when a dialog box really should be modal—but use this only when absolutely necessary. Another good example of an appropriate use for a modal dialog box is when the user wants to open a network connection. Once the request is made, no other activity in the program can be allowed until the dialog box is answered.

Think carefully about whether each of your user dialog boxes should be modal or nonmodal. Most importantly, use dialog boxes only when you need to ask or tell the user something important. Few things annoy an end user more than a barrage of dialog boxes for every little thing when a simple display message will do. On the other hand, urgent, critical messages *should* use dialog boxes. When the user chooses to *quit*, for example, the system should give him or her a chance to cancel that request. Never take drastic action without first confirming it with the user!

7. How to Use Colors in Your GUI

The use of colors in GUIs is a controversial topic. Used correctly, colors can add aesthetic value and provide visual clues as to how to use and interpret an application. Used incorrectly, colors can be visually distracting, irritating, and confusing. In addition, poor color selections can make attractive displays ugly. In general, it is best to design your displays in monochrome and add only small color highlights. More ambitious color designs should be attempted only when you have the time to study the subject thoroughly. Here are some tips that will give your application good, conservative color usage:

- Begin by designing your displays in monochrome.
- Generally backgrounds should use lighter colors than foregrounds.
- When finding colors that work well together, start by choosing the background colors first, and then finding foreground colors to match those.
- Choose colors that are understated rather than bold. Bold colors might make striking first impressions, but they will age quickly and badly. (Remember, you don't want to irritate your end users, or the assessor!)
- Choose just a few colors to accent your application.
- Try to use color to support themes or logical connections.
- Avoid relying on cultural meanings for colors, red may mean "stop" in the Americas, but it has very different meanings in other parts of the world.
- Reds, oranges, and yellows tend to connote action.
- Greens, blues, and purples tend to connote stability.
- When users see different elements of the same color, they will associate those elements to each other, so be careful!

8. How to Test Your GUI

In this section we're going to talk about two different kinds of testing:

- Design testing, which occurs during the design phase of the project
- Code testing, which occurs once the coding phase is complete

There are other distinctions that are often drawn in the arena of testing, unit testing, system testing, regression testing, and so on. We're going to stay at a higher level and discuss design testing and code testing.

Testing Your GUI's Design

In general, the more people you show your GUI design to, the better your ultimate GUI will be. A difficulty in GUI design is that as the designer, you become too close to the application, and it becomes difficult for you to take the perspective of a new user. So, the best way to test a GUI design is to run it by users, let them ask questions, ask them questions, gauge their reactions. Do they seem to use the displays naturally, or do they stumble around looking for the correct sequences? Here are some tips to help you get the most out of your design testing:

- Test your design iteratively, in many stages:
  - Walk through the design when it's on paper.
  - Your paper designs should include use-case flows, and incorporate dialog boxes and warnings.
  - Show users your displays when all that exists are the widgets, with no real logic working.
- Prototype particularly crucial aspects of the application, and do usability testing on those key segments before the rest of the application is complete.
- Get feedback as frequently as possible—you won't be the one using this system, and the people who *will* should have a strong voice in its design.
- The corollary is don't do too much work without getting some feedback. The process should be one of constant refinement, and as such you don't want to invest too much time in a design that the users dislike.
- Make your widgets do the right thing:
  - If you are using a component that can scroll, make it scroll.
  - Avoid using *lists* or *combo boxes* for entries that have extensive ranges. For example, don't use a list for entering a user's year of birth.
  - As a corollary, if the only valid entries for a field come from a list somewhere (like a database), show the user the list if they want to see it (maybe a combo box)—don't make the user guess (for instance, a list of sales territories).

    - When finding and opening a file, use a modal dialog box.
    - Keep your navigation buttons in a well-aligned group.

Testing Your GUI's Code

Testing GUI code can be extremely challenging. By their very nature, GUIs are flexible in their behaviors; even simple GUIs offer billions of possible paths through their features. So how do you test all of these paths? Well, you really can't, but you can hit the high points, and there are approaches that will help you produce a solid application with a finite amount of testing. The key is to approach your testing from several radically different perspectives; the following tips will help you to create a robust and effective test plan:

- One avenue of testing *must* be use-case testing:
    - Have the users run through the system using copies of live work orders and scenarios.
    - If certain scenarios are missing from a set of live work orders, create simulated work orders to test the remaining system features/use-cases.
    - If possible, have the users test the system in parallel with their *live* activities. You will want to create duplicate databases for these parallel tests, and for sure there will be overhead, but parallel testing is a very effective way to test not only for bugs, but also to verify that the system can handle all of the scenarios that the users will run across.
- As the developer, it can be hard to really put your system through its paces, but if you pay attention to your own gut reactions, you can determine those areas where you are afraid to try to break things. Wherever you hesitate to tread, tread heavily.
- Enter invalid data everywhere.
- Test the limits of any data ranges that might exist.
- Force shutdowns of your system at critical stages to determine if transactions are properly encapsulated.

Key Points Summary

Some fun facts to remember when designing and implementing your project's GUI are shown next.

Technical GUI Considerations

- Issues with Swing and JTable
- MVC and why it helps extensibility
- The event-handling model
- Inner classes

Usability Key Points

- Use standard GUI presentation styles.
- Make it easy to learn and easy to use.
- Make it behave as standard GUIs are expected to behave.
- Develop *use-cases* to help define the scope of your GUI's capabilities.
- Document the four phases of each use-case: Find, Display, Verification, Finalization.
- The first several iterations of screen design should be with pencil and paper.

When designing screens, keep the following points in mind:

- Screens should be balanced and clutter-free.
- Elements should be grouped logically.
- Standard workflow should tend to go left to right, top to bottom.
- Action commands should be placed near their logical counterparts.
- Navigation commands should be placed in the menu or tool bars, or at the bottom of the screen (perhaps on the right side).
- Choice of widgets is important—they should match their standard use.
- Your project will probably call for you to use a JTable.
- Try to align your screen elements along invisible vertical and horizontal lines.
- When aligning labels and text fields, right-justify the labels and left-justify the fields—they will converge on a vertical line.
- When designing your menus, keep them as standard and predictable as possible; there are de facto standards that should be respected.

Feedback Principles

- Use short phrases, and positive, short words.
- Minimize the use of jargon and abbreviations.
- Ranges should be described from small to large, now to future.
- Forewarn the user when the system embarks on time-intensive functions. For really slow processes, use a status indicator.
- For the most part your GUI should be nonmodal; consider using dialog boxes when the system becomes, temporarily, modal.

Using Color

- The basic design should be monochrome.
- Backgrounds should be lighter than foregrounds.
- When matching colors, start with the background color.
- Avoid bold colors—they don't age well.
- Choose just a few colors to accent your application, then use them sparingly.
- It's OK to use colors to support themes or logical connections.
- Remember that users will make logical connections when colors match, even if there aren't any logical connections to be made.

Testing Tips

- Include users in the design process.
- Design and review incrementally.
- Consider walkthroughs with no logic behind the widgets.
- Consider walkthroughs of prototypes of key components.
- Test all the use-cases.
- Test in parallel with live systems.
- Focus on testing the areas you are most afraid to test.
- Test with invalid data, and data at the limits of ranges.
- Test by forcing shutdowns at critical stages.

Well, that wraps up our guide to user-friendly GUI design. *Now* all you need to do is learn Swing. Be prepared to spend some time—a lot of time—fiddling with layout managers and the subtleties of JTable. And although you *can* develop Swing applications without really understanding Swing's underlying MVC architecture, you *might* be asked to discuss it in your follow-up essay, so you might as well dig in and learn it all. You won't, however, need to become expert in every single component (widget) in the Swing package. As long as you're familiar enough with all the components to determine which ones best suit your desired behavior (really the *user's* desired behavior), you'll be in good shape for the exam even if you *don't* know anything else about the components you don't use in your final project.