7 Focusing on users and their tasks

The vast majority of software is developed for human beings to use. In this chapter we will show you how to design software for users and how to keep users involved in the process of design. In fact, we will show that if you do not involve users, it will be very hard for you to develop usable software.

In Chapter 4, we showed you the first steps towards involving users in software engineering: one such step is requirements analysis which includes interviewing, brainstorming and use case analysis. In this chapter we will introduce the concept of *user-centered design*. We will also show you how to create and evaluate designs so as to ensure that they are usable – usability is one of the important software qualities that we discussed in Chapter 1.

In this chapter you will learn about the following

- Characteristics of users that every software engineer should understand.
- Various ways of working with users to ensure that a software system has both the required functionality and the required usability.
- Some basic principles for the design of simple graphical user interfaces (GUIs), involving windows, menus, icons and pop-up dialogs.
- How to evaluate user interfaces.
- How to implement basic GUIs in Java.

Users and eXtreme Programming
One of the tenets of the agile method, eXtreme Programming, mentioned in Chapter 1, is that there should at all times be a user representative present and working with the developers.

7.1 User-centered design

During the 50-year history of computing, software developers have often failed to involve users adequately in the development process. For example, during requirements analysis, they have tended to communicate with customers and management, but have often ignored the users. They have then designed, implemented, tested and installed the software, and finally said to the users: 'It's ready to use!' A typical response from the users to this is, 'No, it's not! This lacks functions we need to help us get our work done, it's hard to understand and it's time-consuming to use.' In situations like this, the software is either not used or has to be extensively, and expensively, modified.

User-centered design (UCD) is the term used to describe approaches to software development that focus on the needs of users. Software development approaches that incorporate UCD can help ensure that extensive and expensive modifications are not needed.

Ways to make software development user centered

Many different activities can contribute to making a development process user centered. The following are some of the most important:

- **Understand your users**. Knowing the characteristics of your users allows you to design a system that matches their level of knowledge, their abilities and their preferences. We will discuss this in the next section.
- **Design software based on an understanding of the users' tasks**. Software needs to facilitate the user's work. Performing use case analysis, as discussed in Chapter 4, is the recommended way to ensure this occurs.
- **Ensure users are involved in decision-making processes**. Rather than involving users to a limited extent in requirements analysis, it is better to involve them extensively throughout development. Users cannot be expected to participate in detailed low-level internal design decisions. However, they should be involved in all decision making that relates to the requirements and to the user interface design.
- **Design the user interface following principles of good usability**. Following well-researched UI design principles and guidelines naturally leads you to think about users and their needs. We will study some important usability principles in Section 7.4.
- **Have users work with and give their feedback about prototypes, online help and draft user manuals**. One of the best ways to ensure that users are involved is to develop software iteratively and to involve users in the evaluation of prototypes and user documents. We will look at evaluation of user interface prototypes in Section 7.5.

The importance of focusing on users

User-centered design techniques can significantly improve the quality of the software. They can also reduce the cost to produce, operate and maintain it. Here are some of the clear benefits:

- **Reduced training and support costs**. Large amounts of money are spent both training users to use software and running help desks which support users who have difficulties. If software is designed so that it is more intuitive to use, then its users will need less training and help.
- **Reduced time to learn the system**. Even if users do not take training courses, they still have to invest time to learn how to use the software. The time users spend learning software is a *hidden* cost that cannot be as easily measured as the cost of training courses and help desks. UCD techniques are particularly effective at reducing these hidden costs.
- **Greater efficiency of use**. Another hidden cost is the amount of time that users take to do their work with the system, once they have learned how to use it. UCD techniques can highlight a system's inefficiencies and help to make it faster to use. For example, you might discover that users have to enter unnecessary data, type too many keystrokes, or constantly open and close dialog boxes. Helping users speed up their work can save their employers large amounts of money.
- **Reduced costs by only developing features that are needed**. Without UCD, developers will likely add features that are little used, and hence become 'shelfware'. Development of these features is a waste of money and time. In Chapter 4, we discussed the importance of cutting unnecessary features from requirements; UCD helps you choose which features to keep.
- **Reduced costs associated with changing the system later**. Without UCD, important features will likely be omitted, hence they will have to be added later. If the software is not flexible and maintainable, this is expensive; but even if the missing features can be easily added, the delay will cost money.
- **Better prioritizing of work for iterative development**. UCD techniques permit developers to understand which features should be developed for the first release, and which can be delayed until later. UCD can therefore ensure that the most important features reach the users sooner, and hence the users can start reaping the benefits sooner too.
- **Greater attractiveness of the system means that users will be more willing to buy and use it**. Many systems have *discretionary* patterns of use – users may be able to use competing software or to avoid using any software at all. Benefits from the software can only be achieved if the software is *attractive* to users. UCD techniques can help developers learn what qualities are likely to make users actively choose to use the software.

Exercise

E136 Think of a reasonably complex piece of software with which you have experience (e.g. an operating system, word processor or spreadsheet). Answer the following questions about that system:

(a) Do typical users require training to use this software to its full capacity? Is there anything in the software that could be improved so that less training would be needed? Remember that, as a computing student or professional, you probably have considerable experience with a variety of different software packages; you can therefore figure out a new program much more easily than the average person.

(b) What aspects did you find most difficult to learn when you learned the software? Are there any aspects of the system that you deferred learning because they appeared too complex?

(c) Do you ever find yourself wishing that you could use the software more quickly? What could be improved about the software that would allow you to work faster?

(d) Are there any features that you never use? Do you think that removing the features might make the system easier to use? Or, conversely, do you take comfort in knowing that the features are available, in case you should ever need them?

7.2 Characteristics of users

The first activity that you should perform as part of user-centered design is to understand your users. In Chapter 4, we pointed out that this is something you should start to do in domain analysis, and as the first step of use case analysis.

The following are some of the characteristics that can vary from user to user. As you design software, think about how these characteristics apply to your particular users.

- **Goals for using the system**. Different types of users have different job functions or roles, and therefore have different goals. These goals will lead the users to want different features, and to place different levels of importance on each feature. Understanding goals is critical to defining the problem to be solved, as discussed in Chapter 4, and to choosing an appropriate set of use cases.
- **Potential patterns of use**. For some users, using the system may be optional. The task they would use the system for may not be an essential part of their work, or they may have some alternative way of doing the task. Other users might have no choice but to use the system. Some users may only use the system occasionally – they will therefore have to relearn it every time they want to use it.

- **Demographics**. Demographic variables, such as the age ranges of your users, their educational background, their language and culture, and their geographical location, will all have an influence on the software design. For example, software that is used by adults in their employment will have different characteristics from software that is to be used by children. Special attention will have to be paid if the software is to be used equally by both groups, in the case, for example, of a web search engine.
- **Knowledge of the domain and of computers**. If the users of software are experts in a domain, then the software does not have to provide explanations of concepts and terminology that should be known by those experts. For example, a diagnosis assistant to be used by physicians could assume that its users understand basic anatomy. On the other hand, software that is to be used by the population at large must be significantly simpler to use, providing explanations of all terms that are not in common use. Also, software that is to be used by people who are experienced computer users can be more complex than software that is to be used by people who have little knowledge of computers.
- **Physical ability**. You cannot assume that all users can see and hear. Others may have difficulty using a keyboard or a mouse. You must therefore ensure that the software can interact with devices that help people with disabilities to use it. Some disabilities are quite subtle, such as color-blindness: if you rely on people seeing colors, your application becomes unusable by many people.
- **Psychological traits and emotional feelings**. There are many psychological factors that should be taken into account when designing a system. Many of these relate to the capabilities of human memory and attention (how well humans can focus on a task). For example, a large percentage of the population has to think in order to distinguish right from left – these people (one of the authors included) tend to make mistakes if asked to do so in stressful conditions. Some people have emotional reactions to particular color combinations or imagery. Others might feel a personal attachment to 'the way they used to do things'. Finally, people vary in how easily they get frustrated, or whether they will tend to explore a system out of interest, rather than merely using it to get their task done. We recommend that a psychology course should be part of the basic training of a software engineer.

Exercise

E137 Imagine you are planning to develop the following types of software projects. What different kinds of users should you anticipate? Consider each of the issues mentioned above.

(a) An air-traffic control system.

(b) The GANA GPS-based navigation system discussed in Chapter 4.

Correctly distinguishing left from right in an interface can be critical

On 8 January 1989, British Midland Flight 92 had just taken off from Heathrow, heading towards Belfast. Unfortunately, the *left* engine started emitting smoke and lost power. A Boeing 737 is capable of flying with just one engine. However, the cockpit indicators did not give a strong enough indication about which engine was in trouble; the pilots mistakenly thought the *right* engine had the problem and shut it down instead. Flying with one engine shut down and the other in trouble, the plane crashed while attempting to make an emergency landing. Of the 126 people on board, 47 died. Many people believe that an improved cockpit user interface could have averted this disaster. For more details, see http://pw2.netcom.com/~asapilot/92.html.

(c) A microwave oven.

(d) A payroll system.

7.3 The basics of user interface design

As with many other areas of software engineering, user interface design is the topic of entire books, some of which are listed in the 'For more information' section at the end of this chapter. It has been left out of many general software engineering books because it has historically been seen as a separate discipline. In the early days of computing, user interfaces were much simpler, and the bulk of software design work went into databases and algorithms. In today's world, the user interface is often the most complex part of the system to design. It can take over half of all development effort and is the part that is most likely to cause users to perceive a lack of quality.

User interface design is therefore an essential skill that all software engineers should possess. In this section and the next, we want to highlight some of the most important things that every software developer should know about this topic.

User interface design should be performed in conjunction with other software engineering activities. Prior to UI design, you should have done some domain analysis and made a first attempt at defining the problem. You can then employ use case analysis to help define the tasks that the UI must help the user perform. Next you can begin an iterative process of user interface prototyping in order to address the set of use cases that you have identified. During the prototyping process you will refine both the UI and the use cases. Eventually, results of the prototyping process will enable you to finalize the requirements for the delivered system.

We will present various UI design guidelines in the next section. However, no matter how many guidelines a group of software engineers follow, it would be very arrogant of them to believe they could develop a perfect user interface on their own. The iterative prototyping process must involve extensive discussion with, and evaluation by, both users and other user interface designers. We will discuss techniques to do this in Section 7.5.

Usability versus utility

The overall *usefulness* of a software system can be considered on two quality dimensions:

- Does the system provide the raw functionality to allow the users to achieve their goal? In other words, does it store the right data, allow the right operations and do the right calculations? This quality of a system is often called its *utility*.
- Does the system allow the users to learn and to use the raw capabilities easily? This is *usability*.

Both utility and usability are essential. It is possible to have one without the other, but such a system would be useless. For example, you could imagine a system with very powerful computational capabilities but which is extremely difficult for its users to understand. At the same time, you can imagine a system that is easy to use, but which does not do the correct calculations or store the data a user needs. Both systems would be rejected, for different reasons.

Both utility and usability must be measured in the context of particular types of user (i.e. particular actors). Users with one set of tasks to perform will judge the utility differently from users with different sets of tasks to perform. Also, users with different levels of computer experience and different patterns of use will perceive usability differently. Power users of computers will be able to quickly learn software of considerable complexity, and will then insist that the software allows them to do their job as rapidly as possible. However, users who only occasionally use the software, or are less computer literate, will be more concerned with how easy it is to learn.

Aspects of usability

Usability can be divided into four separate aspects: learnability, efficiency of use, error handling and acceptability.

Learnability is a measure of the speed with which a new user can become proficient with the system. Learnability can be improved in two ways: by having fewer things to learn, or by making the learning process more intuitive. Beginners will perceive a system to be easier to learn if the complex features are hidden from them initially. This can be done by having separate ‘beginner’ and ‘expert’ interfaces. The expert interface might, for example, have additional menu items, fields and buttons. It is common to describe learnability in terms of learning curves, illustrated in Figure 7.1. For example, a user might be able to learn the most important 20 functions of the system in 3 days if the system is simple and intuitive, in 7 days if the system is simple but non-intuitive, and in 11 days if the system is complex and non-intuitive.

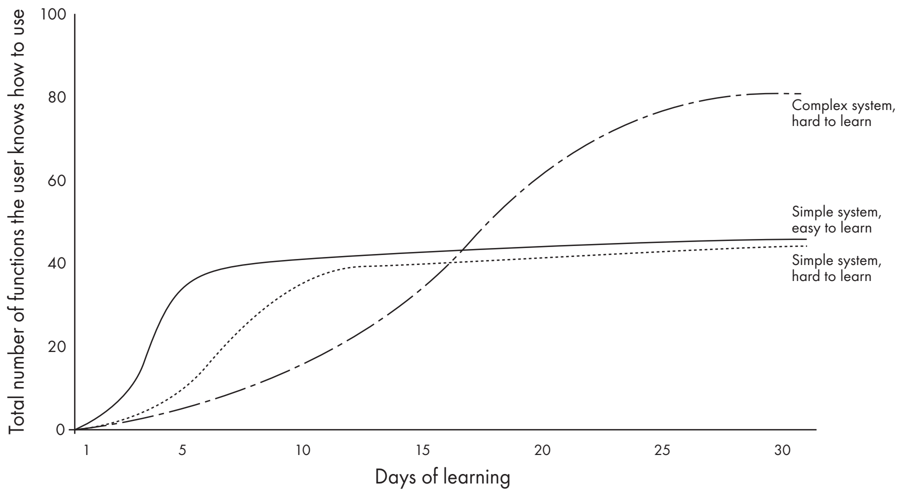


Figure 7.1 **Various learning curves**

Efficiency of use is concerned with how fast an expert user can do his or her work using the system. Suitable metrics for this are: the total number of instances of a small task that a user can do per hour, or the total time required to do a certain large task.

Whereas efficiency of use and learnability consider *ordinary* use, the effectiveness of a system at handling errors is concerned with *abnormal* situations. A system is better at error handling if it *prevents* the user from making errors, if it helps the user to *detect* errors, and if it helps the user to *correct* errors. The following are suitable metrics for these three aspects of error handling:

- Error prevention: Compute the number of error messages that appear per hour of use; a lower number is better.
- Error detection: Count the number of errors a user notices, and divide this by the total number of errors the user makes. Note that some errors, such as navigating to the wrong place, do not result in error messages; these may or may not be noticed by the user.
- Error correction: For each error the user makes, measure the time that elapses from when the user detects the error, to when the user has corrected the error. To compute the percentage of time spent correcting errors, sum the time spent correcting all errors and divide the result by the time spent using the application.

Acceptability measures the extent to which users *like* the system. A system may be learnable and efficient to use, but if users do not like it they will resist using it. Acceptability is a purely subjective phenomenon to which many factors contribute. Various other aspects of usability contribute to it, as does the graphic design.

Basic terminology of user interface design

User interface designers use specific terms that you should understand:

Dialog The word *dialog* (also sometimes spelled *dialogue*) is used generically to describe the back-and-forth interaction between user and computer. The terms dialog or *dialog box* are used to mean a specific window with which a user can interact, other than the main UI window.

Control or widget These words are used interchangeably to describe specific components of a user interface. Typical widgets include menus, lists, input fields and scroll bars.

Affordance The *affordance* is the set of operations that the user can do at any given point in time. Examples of operations include typing into an input field, clicking on a button or selecting an item from a menu. Clicking a button or selecting a menu item are *commands* because they cause the system to perform some computations. UI designers say ‘a button affords clicking’ if clicking on it would cause some action to occur.

State At any stage in the dialog, the system is displaying certain information in certain widgets, and has a certain affordance. Taken together, these are the system’s *user interface state*. The UI state usually changes when the user issues a command. It also changes when the system itself notifies the user of some happening, such as the completion of an earlier command or the arrival of a message.

Mode A *mode* is a situation in which the UI restricts what the user can do – that is, it restricts the affordance. For example, if a dialog appears saying ‘Do you really want to delete a file?’ and all the user can do is click ‘Cancel’ or ‘OK’, then the system is in a mode.

Modal dialog A *modal dialog* is one in which the system is in a very restrictive mode. The user cannot interact with any other window until he or she has dismissed the modal dialog. The most restrictive type of modal dialog has a single ‘OK’ button to dismiss the dialog. A *non-modal dialog* is a separate window with which the user can choose to interact, but is not forced to. Palettes and toolbars are examples of non-modal dialogs.

Feedback Whenever the user does something, the response from the system is called *feedback*. Feedback includes displaying a message, changing a color or displaying a dialog.

Encoding techniques These are ways of representing information so as to communicate it to the user. Tables 7.1 and 7.2 list some of the most common encoding techniques, along with their advantages and disadvantages.

Table 7.1 **Ways of encoding information to be transmitted using sound. Unless backed up by visual cues, these are inaccessible to deaf people**

Medium	*Uses and advantages*	*Problems*
Spoken words	Essential when there is no screen or only a small screen (e.g. a telephone system). Important for blind people who otherwise must rely on tools that convert text into Braille	Can be overheard, violating privacy. Sequential, therefore the user has to request replay if he or she misses a part. Slower for most users than reading text
Music	Can convey mood. Can add attractiveness	Does not usually convey meaning. People have different tastes in music
Abstract sounds (e.g. beeps)	Can give useful feedback about actions that are taking place	Can be hard to interpret
All of the above	Attract attention rapidly at onset, even if the person is not looking at the screen	Can be distracting and annoying

7.4 Usability principles

In this section, we discuss twelve principles that you should apply when designing and evaluating a user interface. After we list the principles, we will give an example of a user interface that violates many of these principles, as well as an improved version of the same system.

The twelve principles

Usability Principle 1: Do not rely only on usability guidelines – always test with users

Each situation is different and there are exceptions to the principles in this section. You should therefore ask the opinions of users and evaluate how they use prototypes. Evaluation is the topic of Section 7.5.

Usability Principle 2: Base UI designs on users' tasks as expressed in use cases

Perform a first iteration of use case analysis and then design the UI based on this. As you evaluate your prototype UI, you will have to go back and revise your use case model as well as your UI.

Usability Principle 3: Ensure the sequences of actions to complete a task correctly are as simple as possible

Make sure users can move from step to step easily as they perform their tasks. You want the user to have to read the smallest amount of text, to navigate the least, to type the least and not to be led into making errors. In particular, make sure the user does not have to select menu items repeatedly to complete a single task. Also, avoid sequences of modal dialogs, since they slow users

Table 7.2 **Ways of encoding information to be transmitted visually. Except for text, which can be spoken or converted to Braille, these are generally inaccessible to blind people**

Medium	*Uses and advantages*	*Problems*
Text written in a language the user can read	Has unlimited ability to express meaning. Simple to generate and display. Accessible by blind people using Braille translators	Takes a lot of space. Writing clearly and unambiguously is hard. Not usable by young children or the illiterate. Hard for users to scan quickly
Fonts (including font family, as well as bold, italics and size attributes)	Add emphasis to text, and reinforce its structure, thus simplifying and highlighting information	Using too many fonts results in confusion and a cluttered appearance. Decorative or unusual fonts can be distracting
Icons (simple and abstract graphics, each representing a specific action or object)	Allow many commands or objects to be listed in less space than is possible with text. Users can scan the screen to find an icon faster than they can scan to find particular text	Notoriously difficult for users to interpret or distinguish. Require artistic skill to create
Diagrams (convey objects and their relationships)	Can communicate or summarize complex concepts or mechanisms more easily than other techniques	Can be hard for users to interact with or interpret. Can be expensive to generate automatically
Photographs and hand-drawn images of reality	Can help users better appreciate reality	Can take a lot of space on screen and can slow response time due to downloading
Animations and video	Provide high impact communication of complex information. Entertaining and hence attractive for users	Bandwidth-intensive, hence reduce response time. Sequential, requiring replay if users miss parts. Users cannot quickly scan them. Expensive to produce. May be annoying
Purely decorative graphics	Make the interface attractive and helps to emphasize its organization	Can be distracting or annoying
Colors	Draw attention to specific items. Convey organization (items colored similarly are related). Makes the UI more attractive. Users can almost instantly notice a small spot of color on the screen	Users cannot distinguish among large numbers of colors. Some color combinations clash. Color-blind people cannot see differences in hue. Some colors (e.g. bright red) can be distracting if overused
Grouping, bordering and organizing in columns or tables	Help to convey the organization of information and reduce its perceived complexity	No problems
Flashing	Rapidly draws attention to items	Distracting and annoying. Fast flashing can cause epileptic seizures and migraine headaches

down and give them the feeling that the computer is in control of the interaction.

Usability Principle 4: Ensure that the user always knows what he or she can and should do next, and what will happen when he or she does it

At any one time there are usually several things that a user can do next – i.e. the system affords several possible actions. Perhaps the user can click on one of several icons, select one of several menu items, or type data into one of several fields. When designing the UI, take note of all these things the user should be able to do; make sure that the user can clearly see how to do all the things that are possible, and 'gray out' those options that are temporarily not available. Make the things the user will want to do most often stand out; they could be larger, in a separate box, or colored more brightly. The consequences of each action should also be clear.

The importance of analyzing the task and interacting with users

One of the reviewers of this book relates the following story. Some designers had put considerable work into developing a new graphical user interface to control an existing piece of hardware more easily. The UI team were excited about the way the system would give the hardware operators easy access to all sorts of information. However, when they showed the system to the users, one of them said, 'Well, I guess that's nice, but all we need to do is press "Start" and "Stop"'.

Usability Principle 5: Provide good feedback, including effective error messages

When a change of state occurs, make sure it is clearly visible to the user. Some specific guidelines are:

- If some operation is taking more than a few seconds, provide a progress bar so that the user knows what is going on.
- Always keep the user informed about where he or she is located among the various windows and pages.
- Communicate clearly to the user when something goes wrong, regardless of whether the problem arises from the user making a mistake, or from a problem with the system itself. Error messages should be informative, telling the user the exact thing that has gone wrong and exactly how to correct the problem, if that is possible.

Example 7.1 *Imagine you are maintaining a program that has to write some data to a specific file. Whenever the program fails to write to the file, for any reason, it currently displays the message: "Error 34 writing file". Describe how this message could be improved.*

1. The message number should not be shown, since this is disconcerting for the user to see.

2. The message should state which file (including which directory) could not be written.
3. The message should tell the user the reason or reasons why it could not write the file. These might include: the existing file is write-protected (for everybody or for specific users), the directory is write-protected, the file system is inaccessible (if it is on a network), the file is locked by another program, there is not enough space to write the file, or the disk appears to be damaged.
4. The message should give the user as much information as possible to help him or her to solve the problem. Such information might include: a) the name and login ID of user who has permission to write to the file or directory, b) the name and process ID of the program that is locking the file, and c) how much space must be freed before there is enough space to write the file.

Usability Principle 6: Ensure that the user can always get out, go back or undo an action

Users will always make mistakes; they will issue incorrect commands or navigate to somewhere they had not intended to go. Therefore you must ensure, where possible, that the user can back out of any action.

In particular, make sure that users can easily undo any operation, even if it has resulted in changes to data. Also make sure that they can easily exit any dialog box and cancel any operation in progress. Providing both these facilities helps users to recover from mistakes, and ensures that they are not afraid of experimenting with the system.

Occasionally it is not possible to undo an action – for example, formatting a disk. If such an action may have serious consequences, you should warn users *before* they perform the action, and ask them to confirm that they really want to do it.

Usability Principle 7: Ensure that response time is adequate

Response time is the time that elapses from when a user issues a command (by selecting a menu item, clicking on an icon etc.) to when the system provides sufficient results that the user can continue his or her work. Response time can be a problem when processing large volumes of information or transmitting data over a network.

Users' perceptions of what is acceptable are determined largely by other applications they use. If your application runs more slowly than users are accustomed to, then users will have a sense that the system is wasting their time.

Operations such as the popping up of menus and echoing of input should appear instantaneous to users. Most other operations should take a second or less, so that the user's train of thought is not interrupted. A few operations may be allowed to take up to about 10 seconds if the user understands that they are naturally time-consuming. An example is loading a complex web page over a slower network connection. Unfortunately, many web sites force users to download excessively large images.

If an operation is to take more than about 10 seconds then warn users in advance. This gives them the opportunity to choose not to perform the operation, and reduces their annoyance with any delay.

When you evaluate a user interface, work on the slowest hardware that end-users are likely to encounter. We suggest assuming that some users will be working with computers that are up to three years old.

Usability Principle 8: Use understandable labels and other encoding techniques

Everything that appears on the screen should be easily understandable to users. This includes all feedback, all elements of the affordance (e.g. buttons) as well as other information for the user. Refer to Tables 7.1 and 7.2 to select encoding techniques. Also:

- Avoid technical jargon and acronyms.
- Employ technical writers to compose text and graphic designers to create graphics.
- Label items so that their meaning is obvious. You can place captions underneath them or provide pop-up labels that appear when the user moves the mouse over them.

Usability Principle 9: Ensure that the UI's appearance is neat and uncluttered

A very common error among UI designers is to provide users with too much to look at. This distracts users, slows them down and makes it harder for them to learn the system. Web pages are especially prone to information overload, particularly with the presence of advertisements.

Messiness of the layout and graphic design can also be distracting, and results in the user taking longer to figure out how to use the interface.

To achieve a neat, uncluttered UI:

- Only display essential information, but provide a way for the user to request additional information.
- Avoid having large numbers of dialogs that each display only a small amount of information: the user may become lost trying to navigate your system.
- Highlight information that belongs together using boxes, colors and fonts. For example, place a box around related items in a form, and use horizontal lines to separate related items in a menu.
- Avoid using too many different colors, fonts or graphics.
- Line up labels and input fields so that users can more quickly read what they have to enter.

Usability Principle 10: Consider the needs of different groups of users

Earlier, we pointed out that there are different types of users, each with their own needs. You should accommodate the needs of the following categories of people:

- *People from different locales*. A locale is an environment where the language, culture, laws, currency and many other factors may be different. Table 7.3 lists some of the things that differ among locales. When designing a UI, it is important to internationalize it, which means ensuring that it can be easily adapted to different locales. Adapting the system to a particular locale is called *localization*.
- *People with disabilities*. People have many kinds of disabilities. To accommodate blind people, ensure that your application works with programs that convert text to Braille or speech. For example, when displaying an image in a web page, use an 'alt' html tag that describes what the image shows. To accommodate deaf people, ensure the system has visual output that conveys the same information as the sound. To accommodate physically disabled people, ensure that your application can interact with software that permits voice input.
- *Beginners versus experts*. In complex applications, provide a simple mode for beginners, and a fully functional mode for experts. The expert mode would have more icons and fields as well as more items in menus.

Also, consider providing a 'preferences' dialog, to enable users to tailor the system to their particular needs.

Localization and internationalization in Java and in operating systems

Operating systems have to display locale-specific information. This means that you can query the operating system to obtain such information when programming a user interface. However, different operating systems store different types of locale information.

Java has its own class called `Locale` that it uses to format numbers, etc. You can use this as a basis for decisions about locale-specific UI features. Java sets the default locale based on what is set in the operating system. However, many of the issues listed in Table 7.3 are not automatically managed by Java.

Java has classes called `Calendar` and `DateFormat` which allow for the use of the calendars of specific cultures, and a class called `TimeZone` which deals with the difference between Universal Time Co-ordinated (UTC or GMT) and local time.

Usability Principle 11: Provide all necessary help

Ensure that users can easily and quickly access relevant and easy-to-understand help about anything they are trying to do. It should be the objective of UI design to make the system good enough such that users will rarely need to access the help, but it is nevertheless essential to have online help as a backup.

Table 7.3 Some types of information that can differ among locales

Locale-dependent feature	*Issues the UI designer needs to be aware of*
Language	Different languages use different amounts of space, different character sets, different fonts, and run in different directions. Employ a skilled technical translator and ensure he or she runs the system in both languages to verify it
Character set and fonts	Unicode can handle most world character sets, but you also have to ensure that appropriate fonts are available
Direction for reading text	Text in some languages runs left to right or top to bottom. Laying out screens so that they can automatically accommodate this is a challenge
Collating sequences (sort order of words)	Some languages order characters with accents or diacritics at the end of an alphabet, whereas others order them as if the accent or diacritic were absent. Often, the sort order for names in phone books is special
The order and components of peoples' names.	Family name comes first in some cultures, last in some, and is non-existent in others. In some cultures, a person's legal name differs from their commonly used name. Salutations such as Mr and Dr vary widely
Currency and format for displaying currencies	An application may use more than one currency at once. The number of decimal places and the magnitude of values may differ widely. The language somebody uses may not correspond to the currency they use
Time zones	Time-zone abbreviations are not used consistently. Daylight savings time starts and ends on different dates in different places, or may not exist at all
Format for dates, times and numbers	There are many ways of writing dates, times and numbers. Even though the international standard is YYYY/MM/DD, this is often not followed
Calendars and holidays	Although international business is based on the Gregorian calendar, the calendars of particular cultures and religions are also used in some places
Formats for phone numbers, addresses, postal codes, credit card numbers, etc.	You should almost never require a specific format for these items since they differ so widely and can change at any time. Allow free-form input of whatever the user wants to type. A common error is not to allow sufficient characters, or not to allow extensions to be recorded for telephone numbers
Laws and business practices	You have to accommodate different ways of calculating taxes, performing accounting, or keeping records. Patents and other regulations might place restrictions on designs
Icons and metaphors	Icons and other encoding techniques can invoke different impressions in people of different cultures

When you develop help, remember that users are often frustrated when they seek help. Be sure, therefore, that the help system does not increase this frustration.

Focus on help that guides the user through the steps of a task. But avoid help that explains all the details at once. Integrate help with the application, making it context sensitive. For example, allow the user to point to some aspect of the UI, or to an error message, and obtain an explanation of it.

Ensure that the help can be easily searched, and that searches retrieve relevant help.

Usability Principle 12: Be consistent

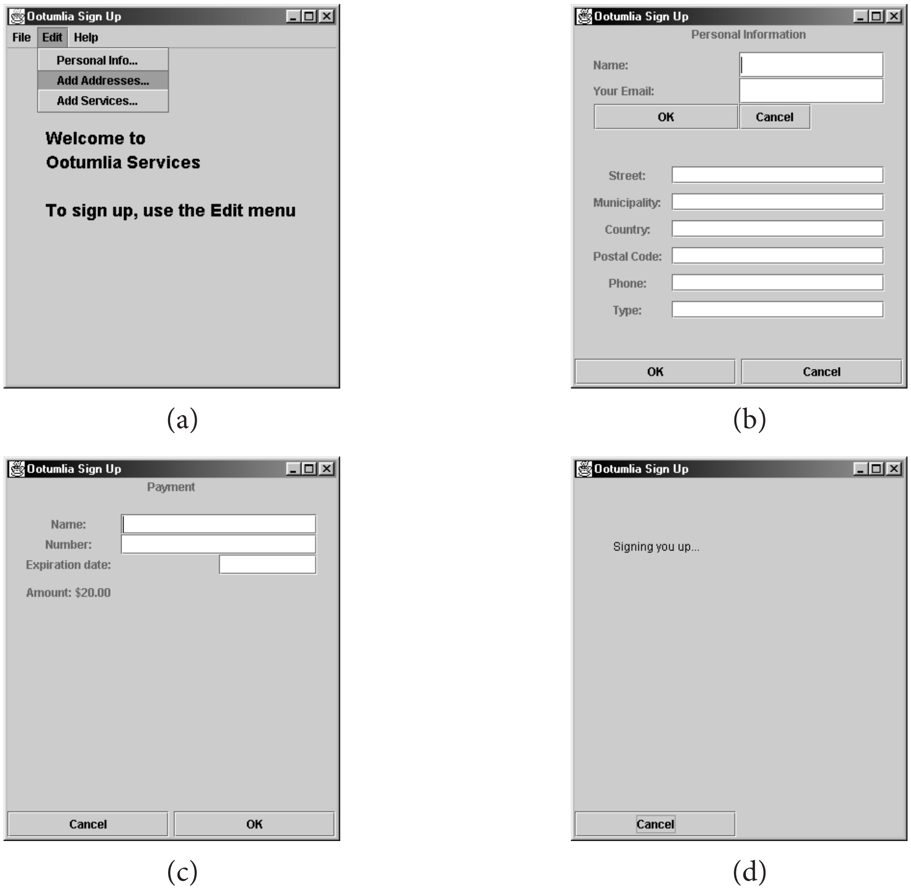
Once users learn how to use one application or dialog, it is a big advantage to them if other applications and dialogs work the same way. Be consistent, therefore, within your own application, make your application follow the standards of the operating system on which it runs, and consider mimicking aspects of other applications. However, ensure that you are not infringing copyrights or patents, and avoid duplicating weaknesses.

An example user interface

Figure 7.2 shows parts of a user interface that has several problems. Figure 7.3 shows the same system after improvements have been made. Before reading on, see how many problems you can find in Figure 7.2.

Here are some of the specific problems found in Figure 7.2, and the corresponding improvements made in Figure 7.3.

- The instructions in Figure 7.2(a), 'To sign up, use the Edit menu' violate Principle 3. Forcing the user to select a menu item in order to start entering information is more complex than is necessary. In Figure 7.3, the user simply has to click on the 'Start' button.
- Although the Edit menu of Figure 7.2(a) shows that the user can add three different kinds of information, there is no indication about the order in which the information should be filled in. Nor is it clear how to finish the sign-up process. These problems violate Principle 4, since the user will not know what to do after completing each step. Figure 7.3, on the other hand, uses a 'wizard' interface: the user can step through the various steps by clicking on 'Next>>', which is prominently visible. Figure 7.3 also numbers each step, which is useful feedback, better conforming to Principle 5.
- In Figure 7.2(a), one of the menu items is 'Add Addresses…'. The intent is to allow the user to specify more than one address (e.g. home, work etc.), but this is not clear. Also, in Figure 7.2(b) there is a field labeled 'type' that is supposed to contain the type of address (home, work, etc.), but the user will probably not understand what to put there. These are both violations of Principles 4 and 8, which require clear instructions and labels. In Figure 7.3, on the other hand, a



Welcome to
Ootumlia Services

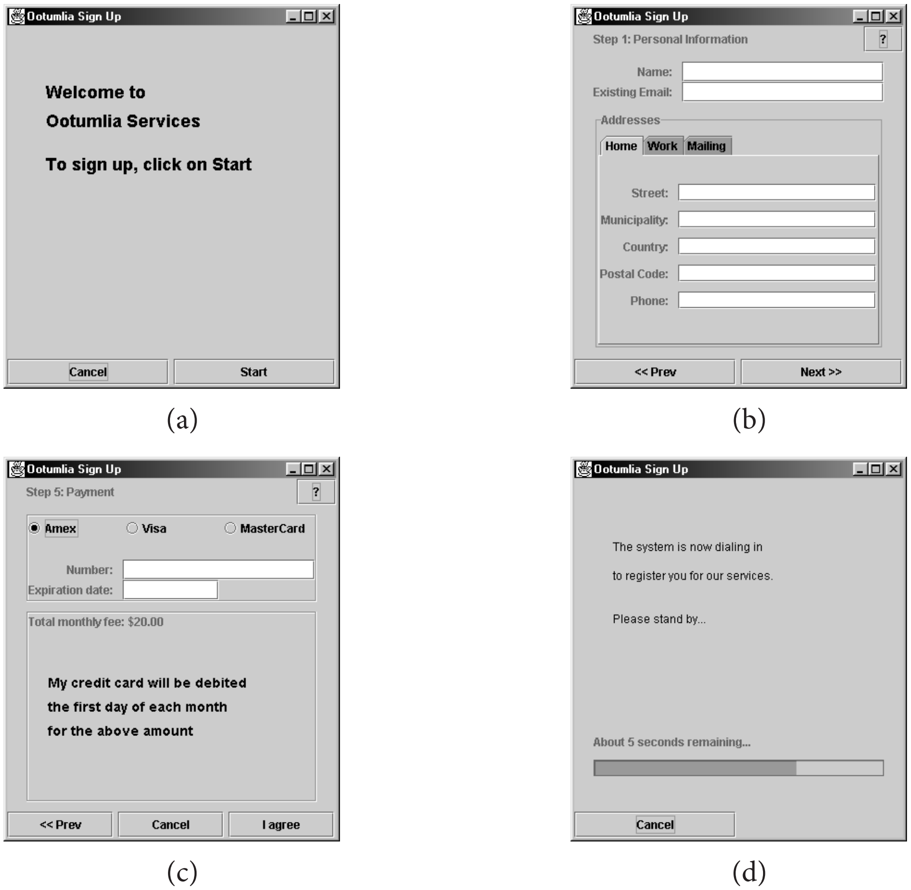
To sign up, use the Edit menu

Signing you up...

Figure 7.2 **Parts of the user interface of an Internet Service Provider sign-up application that has several usability problems**

'tabbed dialog' is used that makes it clearer to the user how to add several addresses.

- In Figure 7.2 there appears to be no way a user can delete a secondary address once he or she has added it. This violates Principle 6, which states that all actions should be undoable. The '<<Prev' button and the tabbed dialog make it much easier for users of Figure 7.3 to change any data they have input.
- Figure 7.2(b) and (c) are modal dialogs, since they only have 'OK' and 'Cancel' buttons. This is another violation of Principle 3. Figure 7.3's 'wizard' interface is preferable.
- Figure 7.2(b) has two sets of 'OK' and 'Cancel' buttons. The user will not understand the reason for this, violating Principles 4 and 8. They also add clutter, violating Principle 9. The duplicate buttons have been removed in Figure 7.3.



Welcome to
Ootumlia Services

To sign up, click on Start

My credit card will be debited
the first day of each month
for the above amount

The system is now dialing in
to register you for our services.

Please stand by...

Figure 7.3 Views of the ISP sign-up application that has improvements compared with Figure 7.2 . It uses a 'wizard' approach as well as tabbed dialogs

- In Figure 7.2(b), the 'OK' button is on the left, and the 'Cancel' button is on the right. This the reverse of what normally appears, violating Principle 12. It is also likely to lead the user to press the 'Cancel' key accidentally, violating Principle 3. In Figure 7.3, the button the user would be most likely to select is always located at the bottom-right corner.
- The layout of Figures 7.2(b) and (c) is messy and inconsistent, violating Principles 9 and 12. In Figure 7.3, all the labels are right justified, and all the input fields line up with each other.
- Figure 7.2(c) has a field labeled 'Name' which is ambiguous and violates Principles 4 and 8. Most users would assume the field is asking for the person's name, but why is system asking for it again? The intent is that the user enters the type of credit card (Visa, MasterCard, etc.) but this is very obscure. Figure 7.3(c) allows the user to select the card type making this much clearer.

- The feedback about the amount owing in Figure 7.3(c) is better than that of Figure 7.2(c), since it is clear that the amount is a *monthly* payment (Principle 5). Also, the user is clearly told that by clicking on 'I agree' he or she is agreeing to payment; it is not clear that this will occur by pressing 'OK' in Figure 7.2(c). This violates Principle 4.
- The feedback provided in Figure 7.2(d) is very weak, violating Principle 5. Figure 7.3(d) gives the user much clearer feedback about what is going on as the system connects to the remote site. The feedback includes a progress bar and an estimate of the number of seconds remaining.
- In Figure 7.2, there is no clear way to access help, violating Principle 11. Figures 7.3(b) and (c) have '?' buttons to provide this capability.

Exercises

E138 Describe the error messages that will need to be displayed in each of the panels shown in Figure 7.3.

E139 You are asked to design the GUI for a software application that can convert audio files from one format to another.

(a) Use the twelve usability principles to draw a paper prototype of this GUI.

(b) Describe how you have adhered to each of the twelve usability principles.

(c) Obtain an application that does the same thing and compare your GUI to this one. There are free applications available on the Internet.

E140 Imagine your goal is to develop a web-based application to access your voice messaging system (or your answering machine). This application can allow you to select messages, play them, drag messages to other applications, etc.

(a) Create a use case model for this application.

(b) Draw the interface of this application. Pay attention to the layout; specify all labels you would use; propose icons or other encoding techniques to make your interface more usable.

E141 Draw a paper prototype of the user interface for the GANA system, whose requirements were presented in Chapter 4. Base your prototype on the textual description of the UI given in the requirements document, and the use cases you developed in Exercise E68(b). You will have to fill in some details not explicit in the textual description, such as the size and shape of buttons etc.

7.5 Evaluating user interfaces

No matter how well a designer adheres to the principles discussed in the last section, usability can only be assured by careful evaluation. In this section, we describe two approaches to evaluation that can be combined to produce a highly usable system.

Evaluation versus quality assurance

In Chapter 10, we will discuss inspection and testing of software – two quality assurance processes. These two processes are used to uncover defects resulting, in general, from violation of requirements.

Heuristic evaluation and usability testing, as discussed here, are analogous to the above but focus on one particular quality: usability. However, there is an important difference: most of the usability defects found do not represent violation of explicit requirements in fact, the recommendations for change resulting from these processes really are *new* requirements – requirements that couldn't be known until you have a system to evaluate. Evaluating usability therefore cannot be left until you think the design and coding are finished.

Heuristic evaluation

Heuristic evaluation involves systematically examining the system, looking for *usability defects* – aspects of the design that might pose problems for users. Heuristic evaluation is the most popular of several techniques that are collectively called *usability inspection*. Usability inspection should be performed on all software; it can be done by regular members of the software engineering team, and by usability experts if they are available.

You can perform a heuristic evaluation of a paper prototype, a finished system, or any intermediate version. It is best to ask two or three people to do each evaluation independently in order to maximize the number of defects found.

Use the following steps when you perform a heuristic evaluation:

1. Pick some use cases to help focus the evaluation. Focus initially on the most important ones.
2. For each window, page or dialog that appears during the execution of the use cases, study it in detail to look for possible usability defects: violations of the principles and guidelines (the heuristics) discussed in the previous section. Be as critical as you can; if you think something has a chance of being a problem for some user, then consider it a defect. It is better to raise a concern about something that is actually not a problem than to ignore something that is.
3. When you discover a usability defect, write down the following information:
   - ❑ A short description of the defect. You may need to include a screen snapshot if the nature of the problem might not be obvious.

❑ Your ideas for how the defect might be fixed.

Your purpose in recording this information is to communicate with other software engineers who will be fixing the defect. You can also learn what to avoid when you next design a user interface.

Exercises

E142 Find an application that performs each of the following tasks. Perform a heuristic evaluation based on each task in order to find the situations where it violates the twelve usability principles described in the last section. Describe each of the defects you find, and suggest how it could be fixed.

(a) A facility for drawing a graph, in a spreadsheet or statistical application. Evaluate changing the graph format (e.g. scatter, bar or line), changing x- and y-axis labels, changing the scale of the axes and adding extra data points.

(b) Facilities for creating a table in a word processor. Evaluate converting text into a table, balancing the widths of columns and making the format of a table look like those in this book.

E143 Download three freeware applications designed to perform the same task. Perform a heuristic evaluation of each. Select the best and worst one and explain why you ranked them this way.

E144 Work in groups of two or three to do the following. First, each member of the group should independently perform a heuristic evaluation of the paper prototype you developed in Exercise E139 or E140. Then the group members should get together and study each other's lists of defects. Determine how many were found by only one person, how many were found by two people and how many were found by all three (if you have three members). This exercise should demonstrate that having more than one evaluator is important.

E145 Perform a heuristic evaluation of the GANA UI that you drew in Exercise E141. Update your GUI if necessary; also, indicate any requirements that should be changed based on your UI review.

Usability testing: evaluation by observation of users

No matter how much work a software engineer puts into user interface design and heuristic evaluation, some usability problems will exist. It is therefore essential to carefully and systematically observe users as they use a prototype of the system, in order to discover these problems.

Some strategies for usability testing include:

- Select users corresponding to each of the most important actors. Remember that a user may have more than one role. Also, try to select people who are both beginners and experts in the domain, as well as people who are experts and non-experts in terms of their experience with computers. You will likely learn different things from observing different types of users; and you will help ensure that the system is suited to different types of users.
- Select the most important use cases for each of the actors you selected in the last step, and determine specific tasks for the users to follow. Each task should be a concrete scenario of one of the use cases.
- Write sufficient instructions about each of the scenarios so that the users know what goal they should try to achieve. Record these on small cards so that you can hand them to the users one at a time.
- Arrange evaluation sessions with users well in advance, and leave plenty of time for each session. Sessions lasting more than an hour are too tiring. A good length is 20–30 minutes. Work with one user at a time.
- At the beginning of the session explain the purpose of the evaluation. In particular, explain to the users that the objective is to evaluate the software, not them. Also, make sure that the users understand that their participation is optional, that they can withdraw at any time, and that whatever happens will be kept confidential.
- Preferably make a video recording of each session. It is very difficult to notice and record all the details of interactions while observing them live. Studying a video later will often bring to light important information. However, video recording can be intimidating to users and may make the logistics of the session harder to manage. If you do record sessions, then test the camera and look at a sample recording in advance to make sure the screen is sufficiently readable. When you look at the recording, you need to be able to understand what the user is doing (you may not be able to completely read all the text on the screen). Also make sure that the sound is clearly audible.
- Converse with the users as they are performing the tasks. Ask them what they are thinking, what they think the system's feedback means, and why they perform various actions. Encourage them to think out loud.
- When the users finish all the tasks, de-brief them. This means ask for their overall impressions and recommendations.
- Analyze any difficulties experienced by the users, no matter how small. There could be times when they had to seek help, times when they made mistakes, or times when they had to think or explore before figuring out what to do.
- Formulate your recommendations for changes to the system that will avoid repetition of the difficulties.

Ethics of usability testing

Whenever you observe users as part of the process of studying software, you need to ensure that you adhere to certain ethical principles. Users may be nervous about participating, may feel an obligation to perform well, may worry about what their manager or others will think about their mistakes, and may become frustrated with the system.

First, ensure that the users fully understand the purpose of the study and are made to feel at ease. They must know that they are volunteers and can stop for any reason. Second, respect their confidentiality: do not involve managers or other people in the process. Furthermore, as soon as your recommendations for changing the UI have been understood by the developers, all records that mention the names of (or show pictures of) individual users should be erased.

If you are performing user studies as part of a *research* activity (i.e. not just for product development), then even stricter ethical guidelines apply. In such situations, users should be asked to sign an 'informed consent' form that clearly specifies their rights, since they are now acting as 'research subjects'.

Exercise

E146 Working in groups of two or three, conduct a usability testing session of some reasonably complicated web site that interests you. You can each take turns being the user.

E147 Download two freeware applications designed to perform the same task and ask two users to use both of them. Follow the usability testing approach described above. Produce a list of recommendations that would improve the usability of each of these applications.

7.6 Implementing a simple GUI in Java

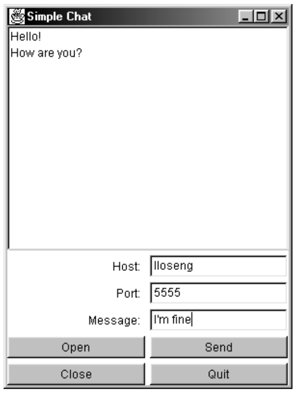
After designing the UI abstractly, and evaluating paper prototypes, it is time to implement it. Java provides two main frameworks for implementing user interface designs: Swing and the Abstract Window Toolkit (also called AWT). AWT is considerably simpler, although rather more limited. Other organizations provide additional frameworks: for example the SWT (Standard Widget Toolkit) is used in the Eclipse environment (www.eclipse.org).

Due to the volume of details that we would have to provide, and the fact that GUI libraries change frequently, we will not discuss how to construct a Swing or SWT-based GUI. However, the basic principles of GUI design in Java remain the same no matter what GUI library you use.

Under the AWT, building a graphical user interface relies on the use of three main elements:

1. `Component`. These are the basic building blocks of any graphical interface. Important subclasses are `Button`, `TextField`, `TextArea`, `List`, `Label`, and `ScrollBar`.
2. `Container`. The role of these is to contain the components that constitute the GUI; the main subclasses are `Frame`, `Dialog` and `Panel`. In Java, a `Frame` is an independent window, with a title and border. A `Dialog` is also a window, but is owned by another window. A `Panel` is designed to be included in another `Container`, even another `Panel`. It therefore also acts as a `Component`. `Panel`s are used to compose complex GUIs.
3. `LayoutManager`. These are classes that define the way `Component`s are laid out in a `Container`. The simplest layout manager is the `GridLayout` that divides a `Container` into a grid where all `Components` occupy equal-sized rectangles. Another useful layout manager is the `BorderLayout`. This time, the `Container` is divided into five areas, designated as Center, North, South, East and West. When a `Container` with a `BorderLayout` is resized, the West and East regions grow vertically only, while the North and South regions grow horizontally only. The Center area grows in both directions.

As an illustration, let us examine how a graphical interface for the SimpleChat program (Phase 1) can be built. The resulting GUI is shown in Figure 7.4.



Hello!
How are you?

lloseng
5555
I'm fine

Figure 7.4 **Simple graphical user interface (GUI) for Phase 1 of SimpleChat**

The first step consists of creating the main window by defining a subclass of `Frame`. All the widgets that compose the window will be attributes of this class.

```
public class ClientGUI extends Frame implements ChatIF
{
  private Button closeB =     new Button("Close");
  private Button openB =      new Button("Open");
  private Button sendB =      new Button("Send");
  private Button quitB =      new Button("Quit");
  private TextField portTxF = new TextField("12345");
  private TextField hostTxF = new TextField("localhost");
  private TextField message = new TextField();
  private Label portLB =      new Label("Port: ", Label.RIGHT);
  private Label hostLB =      new Label("Host: ", Label.RIGHT);
  private Label messageLB =   new Label("Message: ", Label.RIGHT);
  private List messageList =  new List();
```

Note that this class also implements the `ChatIF` interface, as required by the `ChatClient` class.

In order to add a title to the `Frame`, to define its size and to make it visible, the constructor contains the following lines:

```
public ClientGUI(String host, int port)
{
  super("Simple Chat");
  setSize(300,400);
  setVisible(true);
```

The window is composed of a `List`, which is meant to display all the received messages, as well as a series of other widgets located at the bottom of the window. These are laid out using a `BorderLayout`, with the `List` at the Center and the remaining widgets in the South region. However, since only one `Component` can be placed in the South region, a `Panel` called `bottom` is created to contain all the widgets. The following lines, also in the constructor, accomplish this:

```
  setLayout(new BorderLayout(5,5));
  Panel bottom = new Panel();
  add("Center", messageList);
  add("South", bottom);
```

The bottom panel uses a `GridLayout` of 5 lines and 2 columns to lay out its `Components`, as follows:

```
  bottom.setLayout(new GridLayout(5,2,5,5))
  bottom.add(hostLB);
  bottom.add(hostTxF);
  bottom.add(portLB);
  bottom.add(portTxF);
  bottom.add(messageLB);
  bottom.add(message);
  bottom.add(openB);
  bottom.add(sendB);
  bottom.add(closeB);
  bottom.add(quitB);
  ...
}
```

When a message is received, the `ChatClient` instance calls the `display` method of the associated `ChatIF`. The message thus received by the `ClientGUI` is displayed in the `List` widget as follows:

```
public void display(String message)
{
  messageList.add(message);
  messageList.makeVisible(messageList.getItemCount()-1);
}
```

When a user wants to send a message, he or she has to type the message into the `TextField` named `message` and then push the 'Send' button. The `send` method that actually sends the message is written as follows:

```
public void send()
{
  try
  {
    client.sendToServer(message.getText());
  }
  catch (Exception ex)
  {
    messageList.add(ex.toString());
    messageList.makeVisible(messageList.getItemCount()-1);
    messageList.setBackground(Color.yellow);
  }
}
```

We must arrange for the above method to be called whenever the user pushes the 'Send' button. This can be achieved by creating what Java calls an `ActionListener`. `ActionListener` is in fact an interface with only one abstract operation called `actionPerformed`. This operation will be called each time a user pushes the associated button. To associate a `Button` with an `ActionListener` you must proceed as follows (these lines are normally located in the constructor of the `Frame`):

```
sendB.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent e)
  {
    send();
  }
});
```

These lines introduce a feature of Java called an anonymous class – refer to a Java manual for more details on this technique. The basic idea is that we are creating a class which implements the `ActionListener` interface without explicitly naming the new class. We merely give a definition to the abstract method `actionPerformed`, defined in that interface. The `sendB` button is now associated with an `ActionListener` that has an `actionPerformed` method. That method in turn calls the `send` method described earlier. The net effect of this is that when you push the 'Send' button, the `send` method is automatically called.

The complete code for this class and additional examples of AWT GUIs are available on our web site (www.lloseng.com).

7.7 Difficulties and risks in user-centered design

- **Users differ widely**. This means that different types of user will want to perform different use cases. Users will also have different perceptions of what is a good user interface and what is not. In addition, users will want to use the system in different locales.
  Resolution. Make sure you account for differences among users when you design the system. Design it for internationalization. When you perform usability studies, try the system with many different types of users.
- **User interface implementation technology changes rapidly**. Vendors deliver new and changed class libraries for UI design more frequently than they change the core of a programming language.
  Resolution. Stick to simpler UI frameworks that are widely used by others; make sure, however, that they have sufficient power for your needs. Avoid fancy and unusual UI designs, and especially the design of new controls, because they will be more sensitive to changes.
- **User interface design and implementation can often take the majority of the application development effort**. UI design is often thought to be easy and is therefore not explicitly budgeted for as part of the software engineering process.
  Resolution. Make UI design an integral part of the software engineering process, allocating time for many iterations of prototyping and evaluation. Remember that you can save a lot of time by designing the UI using very simple prototypes.
- **Developers often underestimate the weaknesses of a GUI**. Many software developers have not been trained in user interface design. Others tend to feel over-confident about a UI because they personally understand it.
  Resolution. Ensure all software engineers have training in UI development. Always test with users. Study the UIs of any software you use so that you learn to appreciate what constitutes a good and bad UI.

7.8 Summary

In this chapter we have looked at aspects of software development that relate to users: involving users in the development process, and developing user interfaces.

User-centered design is the term given to a variety of software development approaches and techniques that collectively focus on the user. It has been widely recognized that developing software without a user focus results in software that has poor usability.

The USS Vincennes disaster

The *USS Vincennes* was patrolling the Persian Gulf, when operators noticed an aircraft on their combat control system's screen. Unfortunately, in order to determine the altitude and other information about the plane, they had to look at a separate monitor. Under stress, the operators associated the wrong information with the plane they saw. They believed it was a military plane that was threatening them, and shot it down. Actually, it was an Iran Air passenger plane – 290 people lost their lives.

If the designers of the system had followed the principles discussed in this book, this tragedy might never have occurred. If they had performed use case analysis user-centered design, they would have realized that essential information about a plane should be immediately accessible on the combat control system's screen.

User interface design should be performed once you have a clear idea of the use cases. You should follow design principles such as:

- keep the sequences of action simple;
- ensure that the user knows what step to take next;
- provide feedback; ensure response time is adequate;
- use good techniques for encoding information, such as appropriate fonts and colors;
- keep the UI uncluttered;
- ensure the system can be used by different types of users;
- provide help;
- be consistent.

Evaluating user interfaces can be done in two main ways: by inspecting them to verify that they adhere to UI design principles (heuristic evaluation), and by systematically observing users as they use the system (usability testing).

7.9 For more information

The following are some sources of information about user interfaces and usability.

Web sites about usability

- The User Interface Hall of Shame http://digilander.libero.it/chiediloapippo/Engineering/iarchitect/shame.htm is a valuable site including some truly hilarious examples of bad user interface design from real applications. Every software engineer should visit and learn from this site

- useit.com: http://www.useit.com is Jakob Nielsen's personal web site is a very rich source of information related to usability. Jacob Nielsen is probably the best-known author and speaker in the field of user interface design
- Usable Web: http://www.usableweb.com has many links about designing usable web sites
- The Usability Professionals Association: http://www.upassoc.org.

Books about user interfaces and usability

- D. J. Mayhew, *The Usability Engineering Lifecycle: A Practitioner's Handbook for User Interface Design*, Morgan Kaufmann, 1999
- J. Nielsen, *Usability Engineering*, Academic Press Professional, 1994. Discusses techniques for evaluating user interfaces
- J. Nielsen, *Designing Web Usability: The Practice of Simplicity*, New Riders, 2000. Contains numerous color pictures of web sites and large numbers of design guidelines
- J. Johnson, *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan Kaufmann, 2000
- S. Fowler, *GUI Design Handbook*, McGraw Hill, 1997. http://www.books.mcgraw-hill.com/computing/authors/fowler.html
- D. Norman, *The Design of Everyday Things*, Basic Books, 2002

Project exercises

E148 Working in teams of three or four, develop a graphical user interface for the client side of Phase 4 of the SimpleChat system (the version developed in the project exercises of the last chapter). The result of your work will be Phase 5. Use the guidelines presented in this chapter. In particular, undertake the following activities:

(a) Have each person in the team independently create a paper prototype that contains his or her best ideas about how the user interface should appear. Doing this should take no more than about half an hour.

(b) Have each person present his or her paper prototype to the other members of the group. The group members should first point out the good ideas of the prototype, and then give constructive criticism. The creator of the prototype should write down the ideas.

(c) Working together, the group should then take the best ideas from the paper prototypes and develop a new unified paper prototype. Be careful, however, not to add too many features.

(d) Now, using AWT or a GUI building tool, develop Java code to implement your interface, replacing the `ConsoleChat` class.

(e) Evaluate your user interface by arranging for several members of some other group to use it. Take note of any problems they encounter, and update your interface to reduce the problems.

E149 On the book's web site, you will find a class called `DrawPad`. Associated with this there is a class called `StartDraw` that allows you to create an instance of `DrawPad` and draw some simple lines and curves using the mouse. Without changing `DrawPad` at all, add a feature to SimpleChat that will allow you to communicate with other users of SimpleChat by drawing pictures. This would work by having two clients simply open `DrawPad`s and start drawing. The drawing actions should be transmitted as commands via the server. The result of your work will be Phase 6 of SimpleChat.

E150 The SimpleChat system is not internationalized. Its most severe problem is that all the displayed messages are hard-coded in English. Design and implement an internationalized version of SimpleChat that obtains all the displayed messages from a message file. Arrange for the system to determine the locale from the operating system, and then select an appropriate message file. Hint: you can number each message (but do not display numbers to end-users). You will need to develop a way to embed parameters in messages.

E151 Develop a prototype user interface for the Small Hotel Reservation System, as follows. If possible, involve users from the hotel industry.

(a) Do some parallel design of paper prototypes. Then pick the best features of each paper prototype to create a final version.

(b) Review your prototype, following the guidelines discussed in this chapter. Change it as necessary.

(c) Use a rapid prototyping tool to create a partially functional prototype based on your paper prototype. This prototype will not actually manipulate any data and it will not, therefore, make use of the object model you have developed.

(d) Have people use your prototype. Observe the difficulties they have. Change the prototype as necessary.