16 User interface design

Objectives

The objective of this chapter is to introduce some aspects of user interface design that are important for software engineers. When you have read this chapter, you will:

- understand a number of user interface design principles;
- have been introduced to several interaction styles and understand when these are most appropriate;
- understand when to use graphical and textual presentation of information;
- know what is involved in the principal activities in the user interface design process;
- understand usability attributes and have been introduced to different approaches to interface evaluation.

Contents

16.1 Design issues
16.2 The UI design process
16.3 User analysis
16.4 User interface prototyping
16.5 Interface evaluation

Computer system design encompasses a spectrum of activities from hardware design to user interface design. While specialists are often employed for hardware design and for the graphic design of web pages, only large organisations normally employ specialist interface designers for their application software. Therefore, software engineers must often take responsibility for user interface design as well as for the design of the software to implement that interface.

Even when software designers and programmers are competent users of interface implementation technologies, such as Java's Swing classes (Elliott et al., 2002) or XHTML (Musciano and Kennedy, 2002), the user interfaces they develop are often unattractive and inappropriate for their target users. I focus, therefore, on the design process for user interfaces rather than the software that implements these facilities. Because of space limitations, I consider only graphical user interfaces. I don't discuss interfaces that require special (perhaps very simple) displays such as cell phones, DVD players, televisions, copiers and fax machines. Naturally, I can only introduce the topic here and I recommend texts such as those by Dix et al. (Dix, et al., 2004), Weiss (Weiss, 2002) and Shneiderman (Shneiderman, 1998) for more information on user interface design.

Careful user interface design is an essential part of the overall software design process. If a software system is to achieve its full potential, it is essential that its user interface should be designed to match the skills, experience and expectations of its anticipated users. Good user interface design is critical for system dependability. Many so-called 'user errors' are caused by the fact that user interfaces do not consider the capabilities of real users and their working environment. A poorly designed user interface means that users will probably be unable to access some of the system features, will make mistakes and will feel that the system hinders rather than helps them in achieving whatever they are using the system for.

When making user interface design decisions, you should take into account the physical and mental capabilities of the people who use software. I don't have space to discuss human issues in detail here but important factors that you should consider are:

1. People have a limited short-term memory—we can instantaneously remember about seven items of information (Miller, 1957). Therefore, if you present users with too much information at the same time, they may not be able to take it all in.

2. We all make mistakes, especially when we have to handle too much information or are under stress. When systems go wrong and issue warning messages and alarms, this often puts more stress on users, thus increasing the chances that they will make operational errors.

3. We have a diverse range of physical capabilities. Some people see and hear better than others, some people are colour-blind, and some are better than others at physical manipulation. You should not design for your own capabilities and assume that all other users will be able to cope.

Figure 16.1 User interface design principles

Principle	Description
User familiarity	The interface should use terms and concepts drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to recover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system users.

4. We have different interaction preferences. Some people like to work with pictures, others with text. Direct manipulation is natural for some people, but others prefer a style of interaction that is based on issuing commands to the system.

These human factors are the basis for the design principles shown in Figure 16.1. These general principles are applicable to all user interface designs and should normally be instantiated as more detailed design guidelines for specific organisations or types of system. User interface design principles are covered in more detail by Dix, et al. (Dix, et al., 2004). Shneiderman (Shneiderman, 1998) gives a longer list of more specific user interface design guidelines.

The principle of *user familiarity* suggests that users should not be forced to adapt to an interface because it is convenient to implement. The interface should use terms that are familiar to the user, and the objects manipulated by the system should be directly related to the user's working environment. For example, if a system is designed for use by air traffic controllers, the objects manipulated should be aircraft, flight paths, beacons, and so on. Associated operations might be to increase or reduce aircraft speed, adjust heading and change height. The underlying implementation of the interface in terms of files and data structures should be hidden from the end-user.

The principle of *user interface consistency* means that system commands and menus should have the same format, parameters should be passed to all commands in the same way, and command punctuation should be similar. Consistent interfaces reduce user learning time. Knowledge learned in one command or application is therefore applicable in other parts of the system or in related applications.

Interface consistency across applications is also important. As far as possible, commands with similar meanings in different applications should be expressed in

the same way. Errors are often caused when the same keyboard command, such as 'Control-b' means different things in different systems. For example, in the word processor that I normally use, 'Control-b' means embolden text, but in the graphics program that I use to draw diagrams, 'Control-b' means move the selected object behind another object. I make mistakes when using them together and sometimes try to embolden text in a diagram using the key combination. I then get confused when the text disappears behind the enclosing object. You can normally avoid this kind of error if you follow the command key shortcuts defined by the operating system that you use.

This level of consistency is low-level. Interface designers should always try to achieve this in a user interface. Consistency at a higher level is also sometimes desirable. For example, it may be appropriate to support the same operations (print, copy, etc.) on all types of system entities. However, Grudin (Grudin, 1989) points out that complete consistency is neither possible nor desirable. It may be sensible to implement deletion from a desktop by dragging entities into a trash can. It would be awkward to delete text in a word processor in this way.

Unfortunately, the principles of user familiarity and user consistency are sometimes conflicting. Ideally, applications with common features should always use the same commands to access these features. However, this can conflict with user practice when systems are designed to support a particular type of user, such as graphic designers. These users may have evolved their own styles of interactions, terminology and operating conventions. These may clash with the interaction 'standards' that are appropriate to more general applications such as word processors.

The principle of *minimal surprise* is appropriate because people get very irritated when a system behaves in an unexpected way. As a system is used, users build a mental model of how the system works. If an action in one context causes a particular type of change, it is reasonable to expect that the same action in a different context will cause a comparable change. If something completely different happens, the user is both surprised and confused. Interface designers should therefore try to ensure that comparable actions have comparable effects.

Surprises in user interfaces are often the result of the fact that many interfaces are moded. This means that there are several modes of working (e.g., viewing mode and editing mode), and the effect of a command is different depending on the mode. It is very important that, when designing an interface, you include a visual indicator showing the user the current mode.

The principle of *recoverability* is important because users inevitably make mistakes when using a system. The interface design can minimise these mistakes (e.g., using menus means avoids typing mistakes), but mistakes can never be completely eliminated. Consequently, you should include interface facilities that allow users to recover from their mistakes. These can be of three kinds:

1. *Confirmation of destructive actions* If a user specifies an action that is potentially destructive, the system should ask the user to confirm that this is really what is wanted before destroying any information.

2. *The provision of an undo facility* Undo restores the system to a state before the action occurred. Multiple levels of undo are useful because users don't always recognise immediately that a mistake has been made.

3. *Checkpointing* Checkpointing involves saving the state of a system at periodic intervals and allowing the system to restart from the last checkpoint. Then, when mistakes occur, users can go back to a previous state and start again. Many systems now include checkpointing to cope with system failures but, paradoxically, they don't allow system users to use them to recover from their own mistakes.

A related principle is the principle of *user assistance*. Interfaces should have built-in user assistance or help facilities. These should be integrated with the system and should provide different levels of help and advice. Levels should range from basic information on getting started to a full description of system facilities. Help systems should be structured so that users are not overwhelmed with information when they ask for help.

The principle of *user diversity* recognises that, for many interactive systems, there may be different types of users. Some will be casual users who interact occasionally with the system while others may be power users who use the system for several hours each day. Casual users need interfaces that provide guidance, but power users require shortcuts so that they can interact as quickly as possible. Furthermore, users may suffer from disabilities of various types and, if possible, the interface should be adaptable to cope with these. Therefore, you might include facilities to display enlarged text, to replace sound with text, to produce very large buttons and so on. This reflects the notion of Universal Design (UD) (Preiser and Ostoff, 2001), a design philosophy whose goal is to avoid excluding users because of thoughtless design choices.

The principle of recognising user diversity can conflict with the other interface design principles, since some users may prefer very rapid interaction over, for example, user interface consistency. Similarly, the level of user guidance required can be radically different for different users, and it may be impossible to develop support that is suitable for all types of users. You therefore have to make compromises to reconcile the needs of these users.

16.1 Design issues

Before going on to discuss the process of user interface design, I discuss some general design issues that have to be considered by UI designers. Essentially, the designer of a user interface to a computer is faced with two key questions:

1. How should the user interact with the computer system?

2. How should information from the computer system be presented to the user?

A coherent user interface must integrate user interaction and information presentation. This can be difficult because the designer has to find a compromise between the most appropriate styles of interaction and presentation for the application, the background and experience of the system users, and the equipment that is available.

16.1.1 User interaction

User interaction means issuing commands and associated data to the computer system. On early computers, the only way to do this was through a command-line interface, and a special-purpose language was used to communicate with the machine. However, this was geared to expert users and a number of approaches have now evolved that are easier to use. Shneiderman (Shneiderman, 1998) has classified these forms of interaction into five primary styles:

1. *Direct manipulation* The user interacts directly with objects on the screen. Direct manipulation usually involves a pointing device (a mouse, a stylus, a trackball or, on touch screens, a finger) that indicates the object to be manipulated and the action, which specifies what should be done with that object. For example, to delete a file, you may click on an icon representing that file and drag it to a trash can icon.

2. *Menu selection* The user selects a command from a list of possibilities (a menu). The user may also select another screen object by direct manipulation, and the command operates on that object. In this approach, to delete a file, you would select the file icon then select the delete command.

3. *Form fill-in* The user fills in the fields of a form. Some fields may have associated menus, and the form may have action 'buttons' that, when pressed, cause some action to be initiated. You would not normally use this approach to implement the interface to operations such as file deletion. Doing so would involve filling in the name of the file on the form then 'pressing' a delete button.

4. *Command language* The user issues a special command and associated parameters to instruct the system what to do. To delete a file, you would type a delete command with the filename as a parameter.

5. *Natural language* The user issues a command in natural language. This is usually a front end to a command language; the natural language is parsed and translated to system commands. To delete a file, you might type 'delete the file named xxx'.

Each of these styles of interaction has advantages and disadvantages and is best suited to a particular type of application and user (Shneiderman, 1998). Figure 16.2

Figure 16.2 Advantages and disadvantages of interaction styles

Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement Only suitable where there is a visual metaphor for tasks and objects	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users Can become complex if many menu options	Most general-purpose systems
Form fill-in Easy to learn Checkable	Simple data entry	Takes up a lot of screen space Causes problems where user options do not match the form fields	Stock control Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems Command and control systems
Natural language	Accessible to casual users Easily extended	Requires more typing Natural language understanding systems are unreliable	Information retrieval systems

shows the main advantages and disadvantages of these styles and suggests types of applications where they might be used.

Of course, these interaction styles may be mixed, with several styles used in the same application. For example, Microsoft Windows supports direct manipulation of the iconic representation of files and directories, menu-based command selection, and for commands such as configuration commands, the user must fill in a special-purpose form that is presented to them.

In principle, it should be possible to separate the interaction style from the underlying entities that are manipulated through the user interface. This was the basis of the Seeheim model (Pfaff and ten Hagen, 1985) of user interface management. In this model, the presentation of information, the dialogue management and the application are separate. In reality, this model is more of an ideal than practical, but it is certainly possible to have separate interfaces for different classes of users (casual users and experienced users, say) that interact with the same underlying system. This is illustrated in Figure 16.3, which shows a command language interface and a graphical interface to an underlying operating system such as Linux.

Web-based user interfaces are based on the support provided by HTML or XHTML (the page description languages used for web pages) along with languages such as

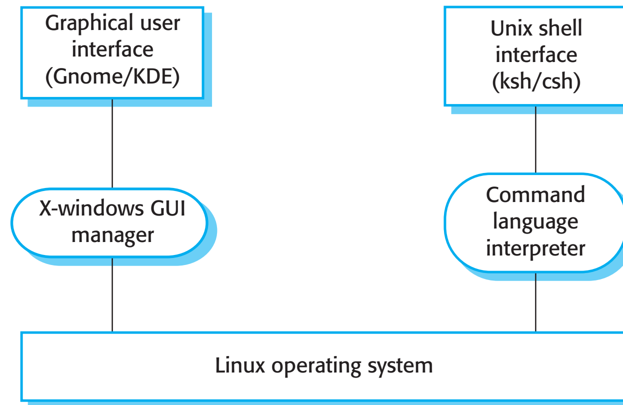
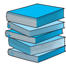


Figure 16.3 Multiple user interfaces

Java, which can associate programs with components on a page. Because these web-based interfaces are usually designed for casual users, they mostly use forms-based interfaces. It is possible to construct direct manipulation interfaces on the web, but this is a complex programming task. Furthermore, because of the range of experience of web users and the fact that they come from many different cultures, it is difficult to establish a user interface metaphor for direct interaction that is universally acceptable.



To illustrate the design of web-based user interaction, I discuss the approach used in the LIBSYS system where users can access documents from other libraries. There are two fundamental operations that need to be supported:

1. *Document search* where users use the search facilities to find the documents that they need
2. *Document request* where users request that the document be delivered to their local machine or server for printing

The LIBSYS user interface is implemented using a web browser, so, given that users must supply information to the system such as the document identifier, their name and their authorisation details, it makes sense to use a forms-based interface. Figure 16.4 shows a possible interface design for the search component of the system.

In forms-based interfaces, the user supplies all of the information required then initiates the action by pressing a button. Forms fields can be menus, free-text input fields or radio buttons. In the LIBSYS example, a user chooses the collection to search from a menu of collections that can be accessed ('All' is the default, meaning search all collections) and types the search phrase into a free-text input field. The user chooses the field of the library record from a menu ('Title' is the default) and selects a radio button to indicate whether the search terms should be adjacent in the record.

LIBSYS: Search

Choose collection: All
Keyword or phrase:
Search using: Title
Adjacent words: ● Yes ○ No
Search | Reset | Cancel

Figure 16.4 A forms-based interface to the LIBSYS system

16.1.2 Information presentation

All interactive systems have to provide some way of presenting information to users. The information presentation may simply be a direct representation of the input information (e.g., text in a word processor) or it may present the information graphically. A good design guideline is to keep the software required for information presentation separate from the information itself. Separating the presentation system from the data allows us to change the representation on the user's screen without having to change the underlying computational system. This is illustrated in Figure 16.5.

The MVC approach (Figure 16.6), first made widely available in Smalltalk (Goldberg and Robson, 1983), is an effective way to support multiple presentations of data. Users can interact with each presentation in a style that is appropriate to the presentation. The data to be displayed is encapsulated in a model object. Each model object may have a number of separate view objects associated with it where each view is a different display representation of the model.

Each view has an associated controller object that handles user input and device interaction. Therefore, a model that represents numeric data may have a view that represents the data as a histogram and a view that presents the data as a table. The model may be edited by changing the values in the table or by lengthening or shortening the bars in the histogram. I discuss this in more detail in Chapter 18, where I explain how you can use the Observer pattern to implement the MVC framework.

To find the best presentation of information, you need to know the users' background and how they use the system. When you are deciding how to present information, you should bear the following questions in mind:

1. Is the user interested in precise information or in the relationships between data values?

2. How quickly do the information values change? Should the change in a value be indicated immediately to the user?

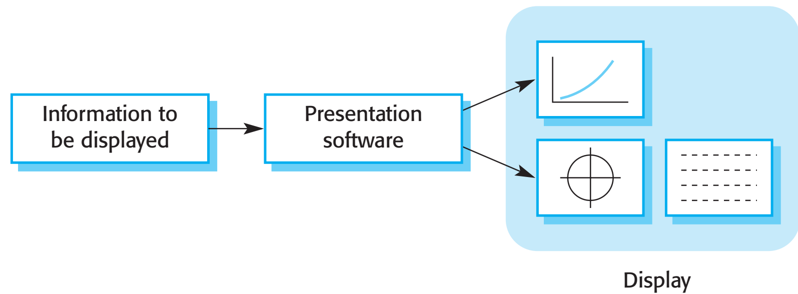


Figure 16.5 Information presentation

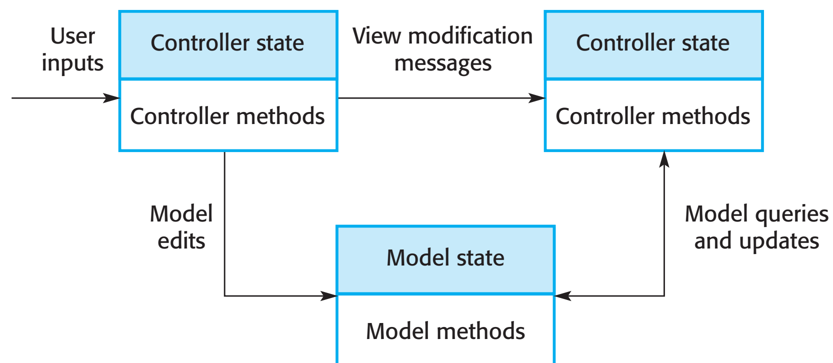


Figure 16.6 The MVC model of user interaction

3. Must the user take some action in response to a change in information?
4. Does the user need to interact with the displayed information via a direct manipulation interface?
5. Is the information to be displayed textual or numeric? Are relative values of information items important?

You should not assume that using graphics makes your display more interesting. Graphics take up valuable screen space (a major issue with portable devices) and can take a long time to download if the user is working over a slow, dial-up connection.

Information that does not change during a session may be presented either graphically or as text depending on the application. Textual presentation takes up less screen space but cannot be read at a glance. You should distinguish information that does not change from dynamic information by using a different presentation style. For example, you could present all static information in a particular font or colour, or you could associate a 'static information' icon with it.

You should use text to present information when precise information is required and the information changes relatively slowly. If the data changes quickly or if the

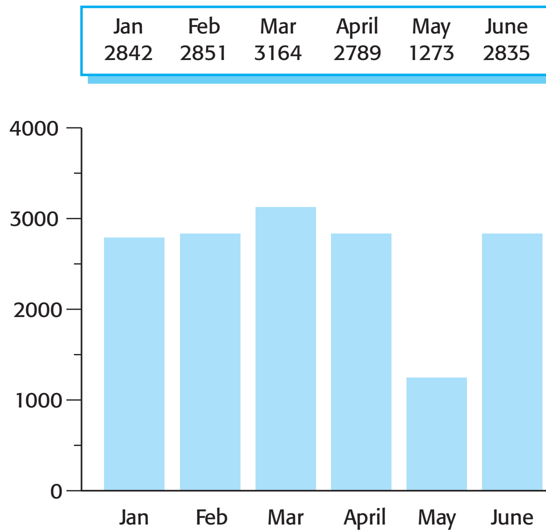


Figure 16.7 Alternative information presentations

relationships between data rather than the precise data values are significant, then you should present the information graphically.

For example, consider a system that records and summarises the sales figures for a company on a monthly basis. Figure 16.7 illustrates how the same information can be presented as text or in a graphical form. Managers studying sales figures are usually more interested in trends or anomalous figures rather than precise values. Graphical presentation of this information, as a histogram, makes the anomalous figures in March and May stand out from the others. Figure 16.7 also illustrates how textual presentation takes less space than a graphical representation of the same information.

In control rooms or instrument panels such as those on a car dashboard, the information that is to be presented represents the state of some other system (e.g., the altitude of an aircraft) and is changing all the time. A constantly changing digital display can be confusing and irritating as readers can't read and assimilate the information before it changes. Such dynamically varying numeric information is therefore best presented graphically using an analogue representation. The graphical display can be supplemented if necessary with a precise digital display. Different ways of presenting dynamic numeric information are shown in Figure 16.8.

Continuous analogue displays give the viewer some sense of relative value. In Figure 16.9, the values of temperature and pressure are approximately the same. However, the graphical display shows that temperature is close to its maximum value whereas pressure has not reached 25% of its maximum. With only a digital value, the viewer must know the maximum values and mentally compute the relative state of the reading. The extra thinking time required can lead to human errors in stressful situations when problems occur and operator displays may be showing abnormal readings.

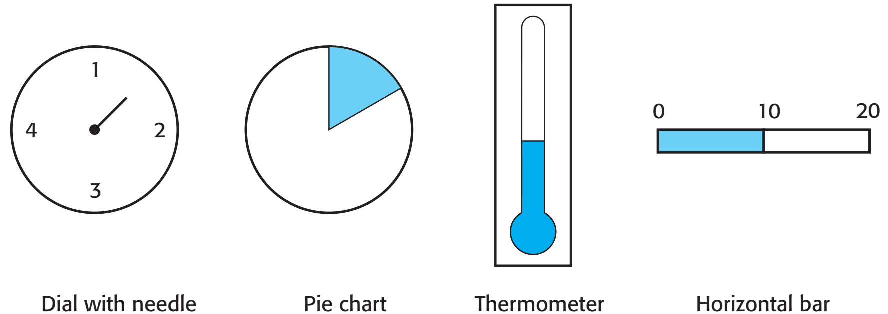


Figure 16.8 Methods of presenting dynamically varying numeric information

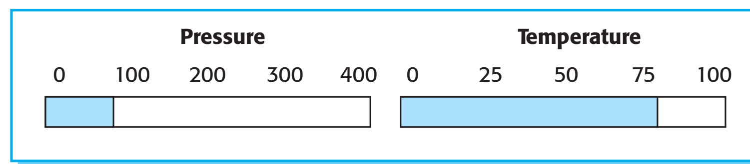


Figure 16.9 Graphical information display showing relative values

When large amounts of information have to be presented, abstract visualisations that link related data items may be used. This can expose relationships that are not obvious from the raw data. You should be aware of the possibilities of visualisation, especially when the system user interface must represent physical entities. Examples of data visualisations are:

1. Weather information, gathered from a number of sources, is shown as a weather map with isobars, weather fronts, and so on.
2. The state of a telephone network is displayed graphically as a linked set of nodes in a network management centre.
3. The state of a chemical plant is visualised by showing pressures and temperatures in a linked set of tanks and pipes.
4. A model of a molecule is displayed and manipulated in three dimensions using a virtual reality system.
5. A set of web pages is displayed as a hyperbolic tree (Lamping et al., 1995).

Shneiderman (Shneiderman, 1998) offers a good overview of approaches to visualisation as well as identifies classes of visualisation that may be used. These include visualising data using two- and three-dimensional presentations and as trees or networks. Most of these are concerned with the display of large amounts of information managed on a computer. However, the most common use of visualisation in user interfaces is to represent some physical structure such as the molecular structure of a new drug, the links in a telecommunications network and so on. Three-

dimensional presentations that may use special virtual reality equipment are particularly effective in product visualisations. Direct manipulation of these visualisations is a very effective way to interact with the data.

In addition to the style of information presentation, you should think carefully about how colour is used in the interface. Colour can improve user interfaces by helping users understand and manage complexity. However, it is easy to misuse colour and to create user interfaces that are visually unattractive and error-prone. Shneiderman gives 14 key guidelines for the effective use of colour in user interfaces. The most important of these are:

1. *Limit the number of colours employed and be conservative how these are used* You should not use more than four or five separate colours in a window and no more than seven in a system interface. If you use too many, or if they are too bright, the display may be confusing. Some users may find masses of colour disturbing and visually tiring. User confusion is also possible if colours are used inconsistently.

2. *Use colour change to show a change in system status* If a display changes colour, this should mean that a significant event has occurred. Thus, in a fuel gauge, you could use a change of colour to indicate that fuel is running low. Colour highlighting is particularly important in complex displays where hundreds of distinct entities may be displayed.

3. *Use colour coding to support the task users are trying to perform* If they have to identify anomalous instances, highlight these instances; if similarities are also to be discovered, highlight these using a different colour.

4. *Use colour coding in a thoughtful and consistent way* If one part of a system displays error messages in red (say), all other parts should do likewise. Red should not be used for anything else. If it is, the user may interpret the red display as an error message.

5. *Be careful about colour pairings* Because of the physiology of the eye, people cannot focus on red and blue simultaneously. Eyestrain is a likely consequence of a red on blue display. Other colour combinations may also be visually disturbing or difficult to read.

In general, you should use colour for highlighting, but you should not associate meanings with particular colours. About 10% of men are colour-blind and may misinterpret the meaning. Human colour perceptions are different, and there are different conventions in different professions about the meaning of particular colours. Users with different backgrounds may unconsciously interpret the same colour in different ways. For example, to a driver, red usually means *danger*. However, to a chemist, red means *hot*.

As well as presenting application information, systems also communicate with users through messages that give information about errors and the system state. A user's first experience of a software system may be when the system presents an

Figure 16.10 Design factors in message wording

Factor	Description
Context	Wherever possible, the messages generated by the system should reflect the current user context. As far as is possible, the system should be aware of what the user is doing and should generate messages that are relevant to their current activity.
Experience	As users become familiar with a system they become irritated by long, 'meaningful' messages. However, beginners find it difficult to understand short, terse statements of a problem. You should provide both types of message and allow the user to control message conciseness.
Skill level	Messages should be tailored to the users' skills as well as their experience. Messages for the different classes of users may be expressed in different ways depending on the terminology that is familiar to the reader.
Style	Messages should be positive rather than negative. They should use the active rather than the passive mode of address. They should never be insulting or try to be funny.
Culture	Wherever possible, the designer of messages should be familiar with the culture of the country where the system is sold. There are distinct cultural differences between Europe, Asia and America. A suitable message for one culture might be unacceptable in another.

error message. Inexperienced users may start work, make an initial error and immediately have to understand the resulting error message. This can be difficult enough for skilled software engineers. It is often impossible for inexperienced or casual system users. Factors that you should take into account when designing system messages are shown in Figure 16.10.

You should anticipate the background and experience of users when designing error messages. For example, say a system user is a nurse in an intensive-care ward in a hospital. Patient monitoring is carried out by a computer system. To view a patient's current state (heart rate, temperature, etc.), the nurse selects 'display' from a menu and inputs the patient's name in the box, as shown in Figure 16.11.

In this case, let's assume that the nurse has misspelled the patient's name and has typed 'MacDonald' instead of 'McDonald'. The system generates an error message. Error messages should always be polite, concise, consistent and constructive. They must not be abusive and should not have associated beeps or other noises that might embarrass the user. Wherever possible, the message should suggest how the error might be corrected. The error message should be linked to a context-sensitive online help system.

Figure 16.12 shows examples of good and bad error messages. The left-hand message is badly designed. It is negative (it accuses the user of making an error), it is not tailored to the user's skill and experience level, and it does not take context

Figure 16.11 An input text box used by a nurse

Please type the patient name in the box then click on OK

Patient name

MacDonald, R.

OK Cancel

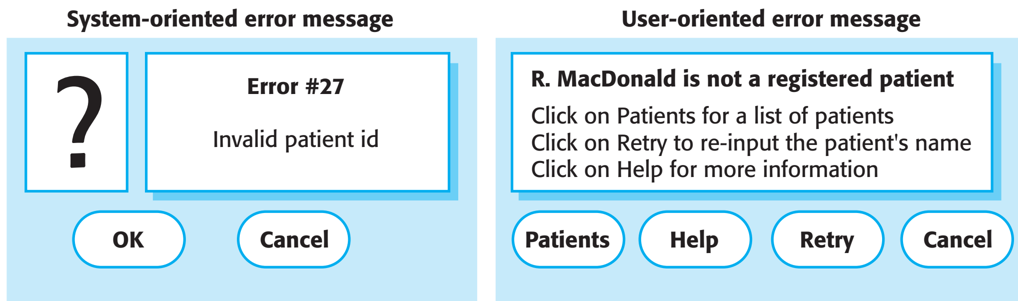


Figure 16.12 System and user-oriented error messages

information into account. It does not suggest how the situation might be rectified. It uses system-specific terms (patient id) rather than user-oriented language. The right-hand message is better. It is positive, implying that the problem is a system rather than a user problem. It identifies the problem in the nurse's terms and offers an easy way to correct the mistake by pressing a single button. The help system is available if required.

16.2 The UI design process

User interface (UI) design is an iterative process where users interact with designers and interface prototypes to decide on the features, organisation and the look and feel of the system user interface. Sometimes, the interface is separately prototyped in parallel with other software engineering activities. More commonly, especially where iterative development is used, the user interface design proceeds incrementally as the software is developed. In both cases, however, before you start programming, you should have developed and, ideally, tested some paper-based designs.

The overall UI design process is illustrated in Figure 16.13. There are three core activities in this process:

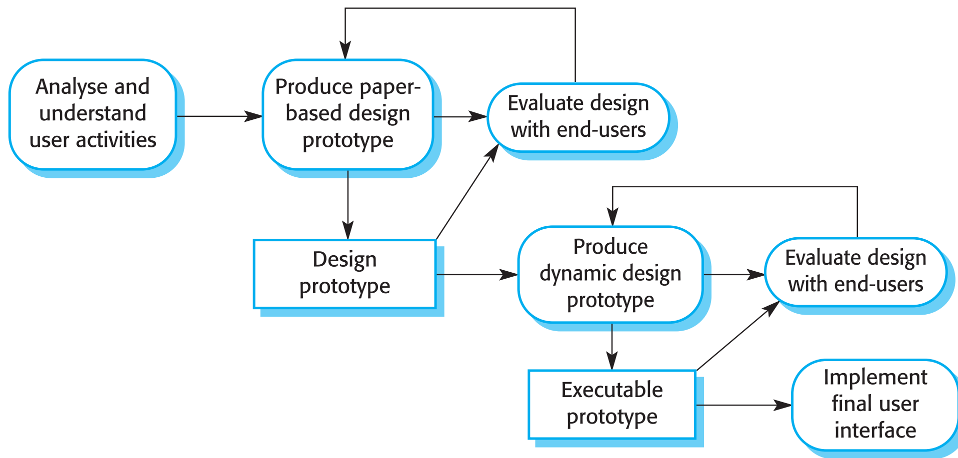


Figure 16.13 The UI design process

1. *User analysis* In the user analysis process, you develop an understanding of the tasks that users do, their working environment, the other systems that they use, how they interact with other people in their work and so on. For products with a diverse range of users, you have to try to develop this understanding through focus groups, trials with potential users and similar exercises.

2. *System prototyping* User interface design and development is an iterative process. Although users may talk about the facilities they need from an interface, it is very difficult for them to be specific until they see something tangible. Therefore, you have to develop prototype systems and expose them to users, who can then guide the evolution of the interface.

3. *Interface evaluation* Although you will obviously have discussions with users during the prototyping process, you should also have a more formalised evaluation activity where you collect information about the users' actual experience with the interface.

I focus on user analysis and interface evaluation in this section with only a brief discussion of specific user interface prototyping techniques. I cover more general issues in prototyping and prototyping techniques in Chapter 17.

The scheduling of UI design within the software process depends, to some extent, on other activities. As I discuss in Chapter 7, prototyping may be used as part of the requirements engineering process and, in this case, it makes sense to start the UI design process at that stage. In iterative processes, discussed in Chapter 17, UI design is integrated with the software development. Like the software itself, the UI may have to be refactored and redesigned during development.

Figure 16.14 A library interaction scenario

> Jane is a religious studies student writing an essay on Indian architecture and how it has been influenced by religious practices. To help her understand this, she would like to access pictures of details on notable buildings but can't find anything in her local library. She approaches the subject librarian to discuss her needs and he suggests search terms that she might use. He also suggests libraries in New Delhi and London that might have this material, so he and Jane log on to the library catalogues and search using these terms. They find some source material and place a request for photocopies of the pictures with architectural details, to be posted directly to Jane.

16.3 User analysis

A critical UI design activity is the analyses of the user activities that are to be supported by the computer system. If you don't understand what users want to do with a system, then you have no realistic prospect of designing an effective user interface. To develop this understanding, you may use techniques such as task analysis, ethnographic studies, user interviews and observations or, commonly, a mixture of all of these.

A challenge for engineers involved in user analysis is to find a way to describe user analyses so that they communicate the essence of the tasks to other designers and to the users themselves. Notations such as UML sequence charts may be able to describe user interactions and are ideal for communicating with software engineers. However, other users may think of these charts as too technical and will not try to understand them. Because it is very important to engage users in the design process, you therefore usually have to develop natural language scenarios to describe user activities.

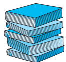


Figure 16.14 is an example of a natural language scenario that might have been developed during the specification and design process for the LIBSYS system. It describes a situation where LIBSYS does not exist and where a student needs to retrieve information from another library. From this scenario, the designer can see a number of requirements:

1. Users might not be aware of appropriate search terms. They may need to access ways of helping them choose search terms.

2. Users have to be able to select collections to search.

3. Users need to be able to carry out searches and request copies of relevant material.

You should not expect user analysis to generate very specific user interface requirements. Normally, the analysis helps you understand the needs and concerns of the

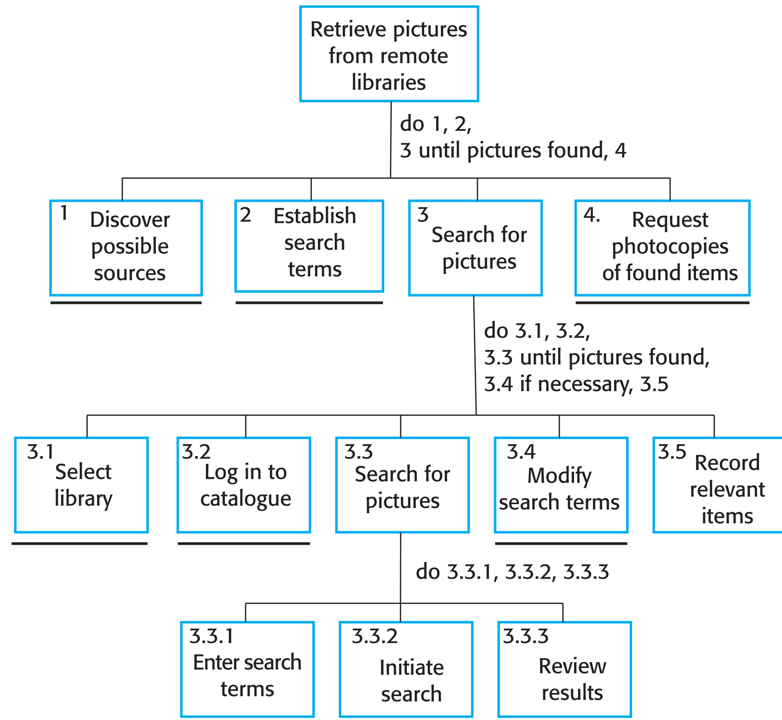


Figure 16.15 Hierarchical task analysis

system users. As you become more aware of how they work, their concerns and their constraints, your design can take these into account. This means that your initial designs (which you will refine through prototyping anyway) are more likely to be acceptable to users and so convince them to become engaged in the process of design refinement.

16.3.1 Analysis techniques

As I suggested in the previous section, there are three basis user analysis techniques: task analysis, interviewing and questionnaires, and ethnography. Task analysis and interviewing focus on the individual and the individual's work, whereas ethnography takes a broader perspective and looks at how people interact with each other, how they arrange their working environment and how they cooperate to solve problems.

There are various forms of task analysis (Diaper, 1989), but the most commonly used is Hierarchical Task Analysis (HTA). HTA was originally developed to help with writing user manuals, but it can also be used to identify what users do to achieve some goal. In HTA, a high-level task is broken down into subtasks, and plans are identified that specify what might happen in a specific situation. Starting with a user goal, you draw a hierarchy showing what has to be done to achieve that goal. Figure

16.15 illustrates this approach using the library scenario introduced in Figure 16.14. In the HTA notation, a line under a box normally indicates that it will not be decomposed into more detailed subtasks.

The advantage of HTA over natural language scenarios is that it forces you to consider each of the tasks and to decide whether these should be decomposed. With natural language scenarios, it is easy to miss important tasks. Scenarios also become long and boring to read if you want to add a lot of detail to them.

The problem with this approach to describing user tasks is that it is best suited to tasks that are sequential processes. The notation becomes awkward when you try to model tasks that involve interleaved or concurrent activities or that involve a very large number of subtasks. Furthermore, HTA does not record why tasks are done in a particular way or constraints on the user processes. You can get a partial view of user activities from HTA, but you need additional information to develop a fuller understanding of the UI design requirements.

Normally, you collect information for HTA through observing and interviewing users. In this interviewing process, you can collect some of this additional information and record it alongside the task analyses. When interviewing to discover what users actually do, you should design interviews so that users can provide any information that they (rather than you) feel is relevant. This means you should not stick rigidly to prepared list of questions. Rather, your questions should be open-ended and should encourage users to tell you why they do things as well as what they actually do.

Interviewing, of course, is not just a way of gathering information for task analysis—it is a general information-gathering technique. You may decide to supplement individual interviews with group interviews or focus groups. The advantage of using focus groups is that users stimulate each other to provide information and may end up discussing different ways that they have developed of using systems.

Task analysis focuses on how individuals work but, of course, most work is actually cooperative. People work together to achieve a goal, and users find it difficult to discuss how this cooperation actually takes place. Therefore, direct observation of how users work and use computer-based systems is an important additional technique of user analysis.

One approach to direct observation that has been used in a wide variety of settings is ethnography (Suchman, 1983; Hughes, et al., 1997; Crabtree, 2003). I discussed ethnography in Chapter 7 as a technique that supports requirements engineering. Ethnographers closely observe how people work, how they interact with others and how features in the workplace are used to support their work. The advantage of ethnography is that the ethnographer can observe intuitive actions and informal collaborations that can then spark further discussions about the work.

As an example of how ethnography can influence user interface design, Figure 16.16 is a fragment from a report of an ethnographic study on air traffic controllers in which I was involved (Bentley, et al., 1992). We were interested in the interface design for a more automated ATC system and we learned two important things from these observations:

Figure 16.16 A report of observations of air traffic control

> Air traffic control involves a number of control 'suites' where the suites controlling adjacent sectors of airspace are physically located next to each other. Flights in a sector are represented by paper strips that are fitted into wooden racks in an order that reflects their position in the sector. If there are not enough slots in the rack (i.e. when the airspace is very busy), controllers spread the strips out on the desk in front of the rack. When we were observing controllers, we noticed that controllers regularly glanced at the strip racks in the adjacent sector. We pointed this out to them and asked them why they did this. They replied that, when the adjacent controller has strips on his or her desk, then this means that a lot of flights will be entering their sector. They therefore tried to increase the speed of aircraft in the sector to 'clear space' for the incoming aircraft.

1. Controllers had to be able to see all flights in a sector (this was why they spread strips out on the desk). Therefore, we should avoid using scrolling displays where flights disappeared off the top or bottom of the display.

2. The interface should have some way of telling controllers how many flights are in adjacent sectors so that controllers can plan their work load.

Checking adjacent sectors was an automatic controller action and it is very likely that they would not have mentioned this in discussions of the ATC process. It was only through direct observation that we discovered these important requirements.

None of these user analysis techniques, on their own, give you a complete picture of what users actually do. They are complementary approaches that you should use together to help you understand what users do and get insights into what might be an appropriate user interface design.

16.4 User interface prototyping

Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing user interface requirements. Evolutionary or exploratory prototyping with end-user involvement is the only practical way to design and develop graphical user interfaces for software systems. Involving the user in the design and development process is an essential aspect of *user-centred design* (Norman and Draper, 1986), a design philosophy for interactive systems.

The aim of prototyping is to allow users to gain direct experience with the interface. Most of us find it difficult to think abstractly about a user interface and to explain exactly what we want. However, when we are presented with examples, it is easy to identify the characteristics that we like and dislike.

Ideally, when you are prototyping a user interface, you should adopt a two-stage prototyping process:

1. Very early in the process, you should develop paper prototypes—mock-ups of screen designs—and walk through these with end-users.
2. You then refine your design and develop increasingly sophisticated automated prototypes, then make them available to users for testing and activity simulation.

Paper prototyping is a cheap and surprisingly effective approach to prototype development (Snyder, 2003). You don't need to develop any executable software and the designs don't have to be drawn to professional standards. You can draw paper versions of the system screens that users interact with and design a set of scenarios describing how the system might be used. As a scenario progresses, you sketch the information that would be displayed and the options available to users.

You then work through these scenarios with users to simulate how the system might be used. This is an effective way to get users' initial reactions to an interface design, the information they need from the system and how they would normally interact with the system.

Alternatively, you can use a storyboarding technique to present the interface design. A *storyboard* is a series of sketches that illustrate a sequence of interactions. This is less hands-on but can be more convenient when presenting the interface proposals to groups rather than individuals.

After initial experiments with a paper prototype, you should implement a software prototype of the interface design. The problem, of course, is that you need to have some system functionality with which the users can interact. If you are prototyping the UI very early in the system development process, this may not be available. To get around this problem, you can use 'Wizard of Oz' prototyping (see the web page for an explanation if you haven't seen the film). In this approach, users interact with what appears to be a computer system, but their inputs are actually channelled to a hidden person who simulates the system's responses. They can do this directly or by using some other system to compute the required responses. In this case, you don't need to have any executable software apart from the proposed user interface.

Further prototyping experiments may then be carried out using either an evolutionary or a throw-away approach. I discuss these approaches to prototyping in Chapter 17, where I also describe a range of techniques that can be used for prototyping and rapid application development. There are three approaches that you can use for user interface prototyping:

1. *Script-driven approach* If you simply need to explore ideas with users, you can use a script-driven approach such as you'd find in Macromedia Director. In this approach, you create screens with visual elements, such as buttons and menus, and associate a script with these elements. When the user interacts with these

screens, the script is executed and the next screen is presented, showing them the results of their actions. There is no application logic involved.

2. *Visual programming languages* Visual programming languages, such as Visual Basic, incorporate a powerful development environment, access to a range of reusable objects and a user-interface development system that allows interfaces to be created quickly, with components and scripts associated with interface objects. I describe visual development systems in Chapter 17.

3. *Internet-based prototyping* These solutions, based on web browsers and languages such as Java, offer a ready-made user interface. You add functionality by associating segments of Java programs with the information to be displayed. These segments (called applets) are executed automatically when the page is loaded into the browser. This approach is a fast way to develop user interface prototypes, but there are inherent restrictions imposed by the browser and the Java security model.

Prototyping is obviously closely associated with interface evaluation. Formal evaluation is unlikely to be cost-effective for early prototypes, so what you are trying to achieve at this stage is a ‘formative evaluation’ where you look for ways in which the interface can be improved. As the prototype becomes more complete, you can use systematic evaluation techniques, as discussed in the following section.

16.5 Interface evaluation

Interface evaluation is the process of assessing the usability of an interface and checking that it meets user requirements. Therefore, it should be part of the normal verification and validation process for software systems. Neilsen (Neilsen, 1993) includes a good chapter on this topic in his book on usability engineering.

Ideally, an evaluation should be conducted against a usability specification based on usability attributes, as shown in Figure 16.17. Metrics for these usability attributes can be devised. For example, in a learnability specification, you might state that an operator who is familiar with the work supported should be able to use 80% of the system functionality after a three-hour training session. However, it is more common to specify usability (if it is specified at all) qualitatively rather than using metrics. You therefore usually have to use your judgement and experience in interface evaluation.

Systematic evaluation of a user interface design can be an expensive process involving cognitive scientists and graphics designers. You may have to design and carry out a statistically significant number of experiments with typical users. You may need to use specially constructed laboratories fitted with monitoring equipment. A user interface evaluation of this kind is economically unrealistic for systems developed by small organisations with limited resources.

Figure 16.17 Usability attributes

Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

There are a number of simpler, less expensive techniques of user interface evaluation that can identify particular user interface design deficiencies:

1. Questionnaires that collect information about what users thought of the interface;
2. Observation of users at work with the system and 'thinking aloud' about how they are trying to use the system to accomplish some task;
3. Video 'snapshots' of typical system use;
4. The inclusion in the software of code which collects information about the most-used facilities and the most common errors.

Surveying users by questionnaire is a relatively cheap way to evaluate an interface. The questions should be precise rather than general. It is no use asking questions such as 'Please comment on the usability of the interface' as the responses will probably vary so much that you won't see any common trend. Rather, specific questions such as 'Please rate the understandability of the error messages on a scale from 1 to 5. A rating of 1 means very clear and 5 means incomprehensible' are better. They are both easier to answer and more likely to provide useful information to improve the interface.

Users should be asked to rate their own experience and background when filling in the questionnaire. This allows the designer to find out whether users from any particular background have problems with the interface. Questionnaires can even be used before any executable system is available if a paper mock-up of the interface is constructed and evaluated.

Observation-based evaluation simply involves watching users as they use a system, looking at the facilities used, the errors made and so on. This can be supplemented by 'think aloud' sessions where users talk about what they are trying to achieve, how they understand the system and how they are trying to use the system to accomplish their objectives.

Relatively low-cost video equipment means that you can record user sessions for later analysis. Complete video analysis is expensive and requires a specially equipped evaluation suite with several cameras focused on the user and on the screen. However, video recording of selected user operations can be helpful in detecting problems. Other evaluation methods must be used to find out which operations cause user difficulties.

Analysis of recordings allows the designer to find out whether the interface requires too much hand movement (a problem with some systems is that users must regularly move their hand from keyboard to mouse) and to see whether unnatural eye movements are necessary. An interface that requires many shifts of focus may mean that the user makes more errors and misses parts of the display.

Instrumenting code to collect usage statistics allows interfaces to be improved in a number of ways. The most common operations can be detected. The interface can be reorganised so that these are the fastest to select. For example, if pop-up or pull-down menus are used, the most frequent operations should be at the top of the

KEY POINTS

- User interface principles covering user familiarity, consistency, minimal surprise, recoverability, user guidance and user diversity help guide the design of user interfaces.
- Styles of interaction with a software system include direct manipulation, menu systems, form fill-in, command languages and natural language.
- Graphical information display should be used when it is intended to present trends and approximate values. Digital display should only be used when precision is required.
- Colour should be used sparingly and consistently in user interfaces. Designers should take account of the fact that a significant number of people are colour-blind.
- The user interface design process includes sub-processes concerned with user analysis, interface prototyping and interface evaluation.
- The aim of user analysis is to sensitise designers to the ways in which users actually work. You should use different techniques—task analysis, interviewing and observation—during user analysis.
- User interface prototype development should be a staged process with early prototypes based on paper versions of the interface that, after initial evaluation and feedback, are used as a basis for automated prototypes.
- The goals of user interface evaluation are to obtain feedback on how a UI design can be improved and to assess whether an interface meets its usability requirements.

menu and destructive operations towards the bottom. Code instrumentation also allows error-prone commands to be detected and modified.

Finally, it is easy to give users a 'gripe' command that they can use to pass messages to the tool designer. This makes users feel that their views are being considered. The interface designer and other engineers can gain rapid feedback about individual problems.

None of these relatively simple approaches to user interface evaluation is foolproof and they are unlikely to detect all user interface problems. However, the techniques can be used with a group of volunteers before a system is released without a large outlay of resources. Many of the worst problems of the user interface design can then be discovered and corrected.

FURTHER READING

Human-Computer Interaction, 3rd ed. A good general text whose strengths are a focus on design issues and cooperative work. (A. Dix, et al., 2004, Prentice Hall.)

Interaction Design. The focus of this book is on designing interaction with computer-based systems. It presents much of the same material as *Human-Computer Interaction* but in a quite different way. Both books are well written and worth reading. (J. Preece, et al., 2002, John Wiley & Sons.)

'Usability Engineering'. This special issue of *IEEE Software* includes a number of articles on usability that have been written specifically for readers with a software engineering background. (*IEEE Software*, 18(1), January 2001.)

EXERCISES

16.1 I suggested in Section 16.1 that the objects manipulated by users should be drawn from their domain rather than from a computer domain. Suggest appropriate objects for the following users and systems.

- A warehouse assistant using an automated parts catalogue
- An airline pilot using an aircraft safety monitoring system
- A manager manipulating a financial database
- A policeman using a patrol car control system

16.2 Suggest situations where it is unwise or impossible to provide a consistent user interface.

16.3 What factors have to be taken into account in the design of a menu-based interface for 'walk-up' systems such as bank ATMs? Write a critical commentary on the interface of an ATM that you use.

16.4 Suggest ways in which the user interface to an e-commerce system such as an online bookstore or music retailer might be adapted for users who have a visual impairment or problems with muscular control.

16.5 Discuss the advantages of graphical information display and suggest four applications where it would be more appropriate to use graphical rather than digital displays of numeric information.

16.6 What are the guidelines that should be followed when using colour in a user interface? Suggest how colour might be used more effectively in the interface of an application system that you use.

16.7 Consider the error messages produced by MS-Windows, Linux, Mac OS or some other operating system. Suggest how these might be improved.

16.8 Write possible interaction scenarios for the following systems:

- Using a web-based theatre booking service to order theatre tickets and pay for them by credit card
- Ordering the same tickets using an interface on a cell phone
- Using a CASE toolset to create an object model of a software system (see Chapters 8 and 14) and generating code automatically from that model.

16.9 Under what circumstances could you use 'Wizard of Oz' prototyping? For what type of systems is this approach unsuitable?

16.10 Design a questionnaire to gather information about the user interface of some tool (such as a word processor) with which you are familiar. If possible, distribute this questionnaire to a number of users and try to evaluate the results. What do these tell you about the user interface design?

16.11 Discuss whether it is ethical to instrument software to monitor its use without telling end-users that their work is being monitored.

16.12 What ethical issues might user interface designers face when trying to reconcile the needs of end-users of a system with the needs of the organisation that is paying for the system to be developed.