Chapter 3. User Centred Design

Table of Contents

Context

In the previous unit you explored various arguments for why user-centred design (or UCD) should be a central feature of interactive systems design. This unit will present an overview of the key UCD tools and practices and show how they relate to conventional systems development processes. UCD tools and techniques differ various ways including their viewpoint (e.g. cognitive psychology or computer science) , formality (e.g. compare evaluations by inspection with formal user modelling for instance), user involvement, when they are applied and deliverables.

Objectives

At the end of this unit you will be able to:

- Identify weaknesses in traditional software development methods in terms of user-centredness.
- Show how UCD methods can improve the design process for interactive computer systems.
- Describe what might make up a UCD toolkit – the methods and tools.
- Distinguish between the various types of UCD tool and technique.
- Know when to apply the tools/techniques

Introduction

In the previous unit we looked in a critical way at interactive systems and their development and did not like much of what we saw. In this unit we begin to look at more positive things: solutions.

There are many tools and techniques for designing systems for improved usability, and the question which should immediately come to mind is ‘which is the best?’ The answer is: ‘it depends’. There are many types of interactive systems and many types of users using them for many different tasks. The initial challenge for a developer wanting to make use of these techniques is to find the most appropriate one for the job at hand.

The first major distinction is between ‘design methods’ and ‘summative evaluation methods’. Essentially the distinction lies in when they applied to the interactive system:

- during the design process, i.e. before there is a tangible system that has been developed, or
- after the majority of the design process has taken place, i.e. when there is an implemented system that can be tested.

There is also a technique known as ‘rapid prototyping’ which aims to occupy the middle ground between design methods and summative evaluation.

We will later present arguments that design methods are preferable, as they should get the major problems out of the system before a costly implementation is produced. Summative methods may identify major problems, but it is likely to be extremely expensive to remedy big problems at this late stage in the design process. Summative methods are still necessary, but should ‘clean up’ small problems with usability such as screen design, while major issues such as fitness for task have been addressed earlier in the design process.

It should be a case of ‘either-or’ for a successful development scheme; to build really usable systems on time and to-budget many of the design and evaluation methods we will describe in this unit will need to be used, but crucially they need to be used well and intelligently by designers. There is not a panacea for usability; no one technique will guarantee a usable system. Many of the techniques that we describe are about identifying usability problems, and structuring the solutions to those problems. How the problems get solved is still left to a great extent to the skill and knowledge of the designers.

Review Question 1

There are two techniques for improving the usability of interactive systems: design techniques and summative evaluation techniques. What differentiates the two? Why, in general, is it impossible to evaluate a system for usability during the design process.

Answer to this question can be found at the end of the chapter.

Activity 1

The task artefact cycle describes a common occurrence; namely that a developer produces a tool for a certain task, and then finds that the users of that artefact use it in a completely different way to that expected.

Think of two examples of where you have used a tool in a way that it really was not designed for, one example being a computer application and one from the wider world.

Now think of the tools/computer programs you use at work or home and see if you can think of ways in which you could use them in different ways to how they were intended.

A discussion on this activity can be found at the end of the chapter.

Traditional Software Development

In this section we look at some ideas and descriptions of how systems get designed. None of these processes of system development take much account of usability, but it is important to understand them so that you can understand the ways in which user centred design can be built into those processes.

Waterfall Development

The waterfall scheme for software development is the classic description of how software is (or should be) developed. There is debate as to how a good a model it actually is of the development process and plenty of variations on its theme have been suggested and described in many software engineering textbooks. Here we shall just concentrate on the classic waterfall model and its implications for user centred design.

Different schemes for development abound but the implications and challenges they set for user centred design are roughly the same.

The waterfall model of software development (see Figure in next section) describes the stages that a development project should pass through from initial contact with the customer and requirements capture, through design and programming to implementation and maintenance.

For the rest of this unit we make the following distinctions: the ‘customer’ is the party who determines what is wanted of the software, and usually pays for it, the ‘users’ are those that actually have to interact with the implemented software and the developers are those that design and implement the software. It is important to realise that the customer need not be the user.

Also note that software is rarely developed on its own, it is usually part of a wider system that is being implemented. Dix et al use the example of aircraft development. The overall lifespan of an

aircraft project can be up to fifty five years, from initial conception to decommissioning. Designing and building software subsystems for an aeroplane is just one part of the many processes that need to carried out to develop and run an aircraft successfully. In this unit we concentrate on the development of software systems, not on overall system development, which is a much wider concern.

In the following description we use the design of a car as an example; it is not a software system, but we use it because it is something familiar.

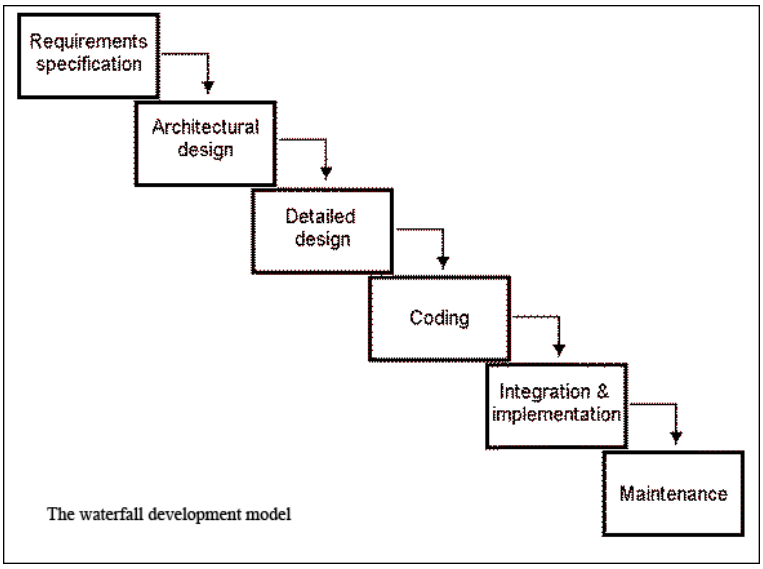
Now we step through each stage in the waterfall model, describing the activities that go on in each stage.

Requirements specification

Requirements describe what a system should do. It is important to distinguish the description of what a system does, from the description of how it does it, which comes later in the cycle.

The developers should liaise with the customers to determine what problem the software is intended to solve or what work it is intended to support. They should also collect information on the environment in which the system is intended to sit, how it relates to any other systems and its user population.

The requirements typically form a contract between the customer and developers. The developers will be obliged to produced a system which does everything that the requirements say it will.



The waterfall development model

Requirements therefore form an interface between customers and developers. The requirements must be understood by the customers, so they realise what they are paying for, and requirements must also be specific and precise enough that developers know what they are supposed to be building. Effectively, to produce a good set of requirements the customer and developers need to be speaking the same language. This may sound a trivial stipulation, but it can be surprisingly difficult to achieve. there are many documented problems where developers produce exactly what is described in the requirements only to find that the requirements do not describe the system that the customer actually wants.

If we were designing a car at this stage in the process we would try to address the problem we are trying to design the car to solve: are we racing the car? or is it a commuter car? or an off-road vehicle? Each of these would give us different requirements for the car: fast, efficient, comfortable, robust, etc. Note that each of these requirements states what is wanted of the car, not how it will be built.

Architectural design

Given a set of requirements the developers can now set themselves to the task of actually producing software. It is a well known software engineering maxim that the effort required to decompose a

problem into lots of smaller problems, solve those small problems and then reassemble those solutions so that they solve the big problem is much less than the effort of solving the big problem in one go.

Based on this principle the architectural design phase decides how to break the problem presented by the requirements into smaller, tractable sub-problems. There may very well be software components available to fulfil some of the sub-problems identified as part of the architectural design phase. Decisions will be made as to whether it is more cost effective to purchase this software or develop it from scratch.

The result of the architectural design phase is the specification of several subsystems that can be developed independently of one another. There will also be a description of how to reassemble these subsystems into the system that satisfies the requirements.

In the case of the car design the car will be split into subsystems: engine, transmission, bodywork, steering etc, with specifications set for each of those subsystems. Those specifications should reflect the overall requirements set in the requirements specification stage. The transmission for an off-road vehicle will have very different specifications to the transmission for a fuel economic car. Also at this stage decisions will be made about buying the components ‘ready made’; will the designers design and build a new engine, or will they buy in an existing make of engine from subcontractors?

Detailed design

If the architectural design phase was conducted well then designers will be able to develop the specified subsystems independently of one another. Indeed in large projects the responsibility for subsystems may be contracted out to other software houses to develop.

For the car, detailed drawings of mechanical components will be produced and detailed assembly guidelines will be put together. At the end of the detailed design phase there should exist an accurate engineering model of the car, from which the car can be built.

Coding

Once the design for a subsystem is completed then programmers can start to produce code. The design phase is now firmly dealing with how a system performs. Once coded the subsystem will be tested to ensure that it fulfils its specification.

There are several techniques for ensuring that correct code is produced. Given that the specification of a subsystem can be expressed mathematically and that program code can be given a mathematical semantics it is possible to mathematically prove that given code is correct to the specification. Doing so is rather arduous, highly skilled and therefore expensive. Only developers of safety critical systems, where correct functioning is absolutely crucial, usually go to these lengths.

Typically a subsystem will be coded and then tested. Generally though it is never possible to fully test a subsystem.

For example, assume that a developer is working on a subsystem that turns off a fuel valve in a boiler when certain conditions occur. The specification will describe the circumstances in which the valve should be shut off: when a certain temperature or pressure is exceeded, or when too rich a fuel mix passes into the boiler. The software subsystem takes input from three sensors which give measurements of temperature, pressure and fuel mix. Let us say that each sensor produces a thousand different possible outputs (a very conservative estimate). Then the subsystem will have a thousand to the power of three (one thousand million) different input configurations. It is simply not possible to test this many configurations, and this example is much simpler than most subsystems developed in the real world.

Mathematically proving that a subsystem fulfils its specification is very arduous, but for all but the simplest subsystems it is impossible to fully test it. Typically subsystems will be tested against crucial inputs (in the case of our boiler it would be tested for the extreme values; the values that move the system from safety into danger, where the valve must work.)

Obviously a car is not coded, but the analogous phase in the design of a car is the production and testing of the individual components that have been designed.

Integration and implementation

Once the subsystems have been coded and tested then they can be assembled into the final system. At this point testing of the overall system will take place and usually the customer is brought back into the process to verify that the emerging system actually matches the needs that they expressed in the requirements. Once the customer verifies this then the system is released to the customer.

Given the manufactured and tested components from the ‘coding’ stage these can now be assembled into a working car. The overall car now exists and can be tested. Of course what is interesting is the extent to which it matches the original requirements set in the first phase.

Maintenance

Once a product is released, work can still continue on it in situ. Indeed it is very unlikely that the developers will have got the system completely right first time and so maintenance needs to take place. It must be noted that the development of a substantial system may take a year or so from requirement capture to implementation, but maintenance may last the working life of the system; twenty years or so. Therefore the major cost of the development actually lies in maintenance. Developing the system carefully may be initially expensive, but should reduce the effort needed in maintenance and therefore pay for itself.

Cars still need servicing and looking after while they are running. Generally this is up to the owner to attend to, but the manufacturer can also be held responsible with warranties and such like.

Discussion of the waterfall model

The waterfall model was developed in the seventies in order to give a framework and identifiable process to software engineering. At this time most software systems were inventory type batch processing systems, with little or no interaction with users.

Now that most systems are interactive, the appropriateness of the waterfall model has been called into question. The waterfall model is intended to develop systems that are primarily functionally correct. Requirements and specifications typically describe what the system should or should not do, and the waterfall model is intended to give developer a process whereby they can produce software that satisfies those requirements.

‘Non-functional requirements’ are not so much about what and how a system operates, but the way in which it operates. Non-functional requirements tended to be treated as of secondary importance to functional requirements. A typically non-functional requirement would be the reliability of a system. If a system did what was required of it, excellent, if it did that without breaking down very often then, that was a bonus.

In the waterfall model usability is considered a non-functional requirement and is therefore given a secondary importance, but in an interactive system usability should be considered as important as function correctness. (Which is not to downplay the importance of functional correctness – the most unusable systems are ones that do not work.)

Later we will briefly look at attempts to tag usability onto the waterfall model, by adding an ‘interface design’ phase. We argue that this is insufficient for genuine usability to ensured and that a concern for usability needs to be addressed at all stages in the waterfall process.

Another interesting consequence of the waterfall model is the cost of making mistakes. If you make a mistake somewhere in the process you usually need to cycle back in the process by going back to the phase in which the mistake was made, correcting it and then continuing through the phases from there. As a rule of thumb it is usually cheap to rectify a mistake if it is spotted soon after it was made.

Hence if a mistake is made in the coding and it is spotted in the implementation phase then it is usually quite cheap to remedy. If, however a mistake is made in the requirements and it is not spotted until implementation then, in effect, you have to rebuild the whole system, which is very expensive. It is therefore absolutely crucial that the correct requirements are gathered, because the customer is only brought back into the process at implementation. If they identify a mistake then, or, more typically, realise that the system produced is not actually the system they really wanted, it is very expensive to remedy these problems.

The waterfall process is a structure to describe how developers should go about designing systems. The concept of design itself is interesting, and has its own particular terminology. In the next section we describe design and some of its terminology.

Designing

Design is about finding solutions to problems. Good design is about finding solutions that solve problems well. Given any problem there will be a myriad of different ways of solving it. These different ways are collectively called the ‘design space’. It is up to the designer to make a choice about which solution in the design space is best suited to the problem. A single decision is called a ‘design step’.

Also important in design is the notion of ‘discharging’ a design ‘obligation’. As a designer you are presented with a statement of a problem, which it is your job to solve. This statement of a problem is known as the obligation, (because you are obligated to solve it) and once you have solved a problem then you are said to have ‘discharged’ that obligation. So a design step is about devising a solution to a problem, and the step is held to be discharged successfully if the designer can demonstrate that their solution solves the problem.

Good design practice tends to put big decisions off as long as possible. If you think of the process of design as being the gradual narrowing of the design space until a single solution is arrived at, then a big decision at the beginning of the process will narrow the design space considerably and leave little scope for design decisions to be made later. ‘Never do today what you can put off until tomorrow’ is a popular and only half jokey maxim of good design.

Now if we think simplistically then for any problem there is a design space, and that design space can be partitioned into solutions which are usable, and solutions that are not. We obviously want to encourage designers to always choose a solution from the usable ones. The following approaches to user centred design are all about helping designers identify which solutions in the design space are usable and which are not.

Things are not that straight forward of course. Usability decisions are unlikely to be clear cut. So the following approaches sometimes also suggest what a designer should do in case of ambiguity and conflict.

The waterfall process is very rigorous and structured. In the next two sections we look at much less rigorous processes, in fact they are so unstructured it is difficult to describe them as processes at all. They are more rough approaches to development, or styles of development. While they are not structured or formal, they are quite common in the real world.

Review Question 2

Why is it cheaper to make mistakes and rectify them early in the waterfall design process?

Answer to this question can be found at the end of the chapter.

Review Question 3

What distinguishes a design step that is good from one that is correct?

Answer to this question can be found at the end of the chapter.

Evolutionary Development

Whereas the waterfall model has a beginning, middle and end, evolutionary development has no such concepts. A system exists and is added to, modified and tinkered with over time to improve it.

The motivation for the improvements may come from different places: new advanced technology may become available that the maintainers of the system may want to use, or new ideas about how the system can be, or is being used, come to light and are incorporated into the system.

Typically innovations are implemented in two ways: firstly small improvements will be added to the working system without taking the system off line. Secondly, radical overhauls to the system may be decided on. In this case the old system will be left working in situ, while the improved system is built and tested concurrently. When the maintainers are happy that the new system is working well then the old system is phased out and replaced with the new.

A good example of evolutionary development is that of digital library systems.

Traditional paper libraries have been around for a very long time. Gradually automation is taking them over. Catalogues have been automated for a long time, and users have been able to electronically search to see if a certain book or journal is in the library, see whether it is on loan and find out which shelf it is housed on. Internet technology means that such catalogues can now be viewed by users at distance. A user can sit at their terminal and see if a book is in the library without having to go there to find out. Then the ability to order books remotely was added to the catalogue systems; the user need never leave their terminals, they could order books from the library and have them delivered to their desk.

Recently publishers have begun to produce journals and books in electronic as well as paper format, and distribute them on CD-ROM and DVD-ROMs. Users can search those media in the library and obtain electronic copies of the books or journal articles they want. Electronic documents can be delivered over the Internet, so the user can simply sit at their terminals and read documents straight away.

The waterfall model is not really sufficient for describing this process. Requirements change with the possibilities offered by the system. A developer designing an electronic catalogue system ten years ago could not really have foreseen the possibilities for libraries that have opened up with the Internet. It is safe to assume that the developers of current digital library systems cannot predict the way that information delivery will change over the next ten years either.

Revolutionary Development

Lastly we look at revolutionary development. There is no set process to a revolution (by definition) and revolutionary software pops up now and again and changes the way we think about computerised systems.

The classic example are spreadsheet programs. The first was developed in the late seventies by a business student fed up of having to perform repetitious calculations. He developed an ‘electronic worksheet’ that did all the work for him. The concept seems obvious and simple now, but it revolutionised the way that many accountants work.

Revolutionary products come from a ‘eureka’ idea: a burst of creative thought that people are famously better than computers at producing. Eureka moments are not understood to any useful degree by psychologists and are therefore not really supportable or predictable.

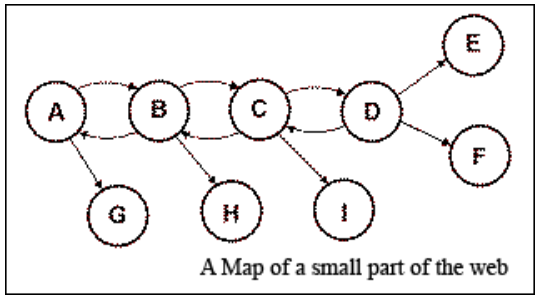
It is salutary to note that though a lot of developers spend their time trying to formulate the next ‘big thing’, the number of genuinely revolutionary computer products in the past twenty years number about five, whereas as the failures that tried to revolutionise are legion.

Review Question 4

Which of the cycles (applied science or task artefact) we described in the opening to this unit best captures evolutionary development?

Answer to this question can be found at the end of the chapter.

Activity 2



A Map of a small part of the web

Recall the analysis of web browsing you performed in unit 2. Now we shall proceed to use that analysis to redesign the web browser, hopefully in a more user centred way.

From unit 2 activity 6 should have resulted in a list of things that you actually did with your web browser, and activity 7 should have resulted in a list of things that your web browser lets you do. This activity compares the two.

Go through the list of browser functionality from unit 2 activity 7 marking off which of those functions you actually used in any of the tasks from activity 6. Apply the following scoring system: if you did not use a function score it 0, if you used it once or twice score it 1, if you used it several times (three to nine times) score it 2, if you used it a lot (over ten times) score it 3.

Now step back. How much functionality scored 0? Most of it? How many functions scored 3? I would guess that the Back' button scored a 3, and maybe the bookmark menu, if you use bookmarks. (Some users do not use bookmarks at all and so they would score a 0).

Now conduct a similar analysis, but this time score each function just with a single task from unit 2 activity 4. So score each function for searching, and then for browsing, and for any other task that you set your self. Now compare the scores for each different task. Do you use different functions for different tasks? I use my bookmarks a lot when searching and the Back button a lot when browsing, but not vice versa.

If you studied other users compile scores for each of them and see how they differ from your scores.

You may have completely different results to me, but one thing should stand out most of the functionality offered by the browser is not used. In other words web browsers are not very well suited to the tasks to which users put them.

In particular lets look at the Back button: most users make great use of it, but can you explain what it actually does? Consider the map of a small corner of the web shown in figure 4. Each circle represents a web page and each arrow a link from one page to another. Imagine you start at page A and jump to B. On B you notice that there is a link to page H that looks interesting, but you first want to look at page C, so you jump there and then jump to page D. Now you decide that you want look at page H, so you need to backtrack to page B, this you do and then jump to page H. Now you press the Back button twice, where do you finish up? Page A or page C? Try and justify your answer.

So we have an even more damning argument for the usability of the browser; not only is most of its functionality unused, what the most used part of the functionality: the Back button actually does, is not very clear.

In summary we have shown that there is considerable scope for redesigning a web browser. This is what we shall look at in activity 4.

A discussion on this activity can be found at the end of the chapter.

User Centred Design

So far we have said little or nothing about usability. The previous section described how software systems get built, but not how usable software systems get built. What would be appealing to software developers would be a new box in the waterfall model called ‘design for usability’ in which the software or designs they have so far developed are passed along to a set of usability experts, who perform their own special rituals to the designs and pass them back into the process, guaranteed to result in improved usability.

The idea of user interface management systems (UIMS for short) is one way of appealing to this idea. UIMS assume that functionality can be separated away from the interface, so software developers can get on with developing their functionality while not worrying about how users make use of that functionality, while interface experts beaver away at designing interfaces that make that functionality easy to use.

Its a beguiling idea, but misguided. Consider the following example:

The Mini is a classic small car, revolutionary in its day and affectionately remembered by most people who owned one. Early models were very spartan in design and manufacture. The speedometer, fuel gauge and other indicators were housed together in a single circular unit which was placed centrally on the dashboard. (Whether this was to make production costs cheaper because the unit stayed in the same place no matter if the car was left or right hand drive is not clear, but it seems likely.) Unfortunately if a moderately tall driver were to use the car they would find that the hand gripping the steering wheel when the car was moving straight forward almost entirely blocked the view of the speedometer and other gauges. To see the gauges the driver would have to take his hand off the steering wheel; not a recommended activity.

Now given that design, the car’s manufacturers could have brought in the most skilled designers to improve the readability of the gauges, but it would not have made the slightest difference. The most well designed gauges in the world are useless if they cannot be easily seen.

The parallel for usability design is clear. If the functionality is inadequate or inappropriate for a given task, the most well designed interface is not going to mitigate that fact at all. Good user interfaces can make good functionality usable, but the functionality needs to be designed for the user too. Therefore user centred design needs to permeate through all the steps of the waterfall model.

The following is very important and should be taken to heart by all usability experts:

Good usability takes more than a good user interface.

Note however that the equating of usability with interface design is deeply ingrained. Even one of the recommended texts for this course seems to make this fundamental error from the outset: Shneiderman’s ‘Designing the user interface’. (This not to say that there is not a wealth of useful information in Shneiderman’s book, but it has an awful title.)

Guidelines

A guideline is a rule about designing interactive systems. They range from very general (‘Keep the user informed about what the user is doing’ from Nielsen 1993) to the very specific (‘Use white space between long groups of controls on menus or in short groups when screen real estate is not an issue’ from the Open Look style guide).

Shneiderman’s ‘Designing the user interface’ is to a large extent a collection of design guidelines, but explanations are included to show why those guidelines are the way they are. (For example look at box 7.2 on page 264 which gives a list of form filling guidelines and then the text below which explains the reason for those guidelines.)

There are two components to a guideline: the guideline itself and the rational behind it. The ultimate idea being that if the guidelines are sound and unambiguous enough then they could be presented to

designers without the rationale behind them. The designers could then apply the guidelines, almost automatically, and be sure of coming up with an usable system.

Such an approach has precedence in all sort of situations; we use sets of rules and guidelines without needing to know the theory that underlies them frequently. A cookbook describes how to make an omelette and if its rules are followed then a perfectly good omelette will be arrived at. The cook does not need to know the science of what happens when the proteins in the egg white are heated in order to successfully cook an omelette.

Rationales for guidelines come from different places and can therefore give the guidelines different levels of authority. Much academic study has attempted to link guidelines to sound scientific theory and if such a link can be established then the guideline will have a considerable level of authority. Other sets of guidelines are based around common sense thinking or possibility previous experience; a practitioner will write down what worked for his design, possibly with some explanation of why.

It is important to distinguish between ‘standards’ which we discussed in the previous unit and the guidelines we are discussing here. They are both rules about what designers should or should not do, but standards carry much more authority; they have to be obeyed. Guidelines are much looser and are often transgressed. Because of the higher authority of standards then they must have a well defined and inspectable rationale behind them.

Problems with guidelines

Unfortunately the science behind usability is not sound enough that this sort of reliance on guidelines can be usefully achieved. The designer will in many cases need to understand, to some extent, the science or rationale underlying the guidelines so that they can be intelligently applied. In many cases usability guidelines contradict one another, so it is important to understand their rationale in order to decide which is the most appropriate.

Usability Engineering

Usability engineers explicitly set down criteria for their designs and then describe measurement schemes by which those criteria can be judged.

An example: Advanced mobile phones pose interesting HCI challenges. Mobile phone manufacturers want to cram as much functionality into their products as possible, but the physical size of the phones and their displays means that normal features for accessing lots of functionality: menus, dialogue boxes, etc., tend to be inappropriate. Assume you have to design a way of accessing phone functionality and you have decided on a restricted menu system. Menus must not be long, perhaps a maximum of six items per menu. To compensate for this you may decide to nest menus, but if you do this then it is easy for the user to forget whereabouts in the menu hierarchy they are, so feedback must given.

As a usability engineer you set yourself the explicit goal of avoiding ‘menu lostness’; the extent to which the user gets irredeemably lost in a menu hierarchy. On each menu there is the ability to ‘bail out’ and return to the highest level menu. If a user bails out like this without invoking some functionality then we assume that they have failed to find what they are looking for and become lost.

The usability specification for this aspect of the menu system may look like the following:

Attribute	Menu lostness
Measuring concept	success in finding and invoking the desired functionality
Measuring method	The ratio of successfully found functionality to bail outs
Now level	This is a new product: there is no recorded now level
Worst case	50% of menu use results in bail outs

Planned level	10% of menu use results in bail outs
Best case	No bail outs

The attribute ‘menu lostness’ describes what is wanted of the system in usability terms. (Note though that ‘menu lostness’ is a bit of an opaque term. There should be accompanying documentation described exactly what it means.) The measuring concept describes what we are looking for in a system in order to judge whether the attribute is fulfilled or not, and the measuring method describes how we should go about looking for it. The now level describes the current state of the system; how the current system fares against the measurements. The worst case, planned level and best case describe what the designer should aim for in their design.

Note that like good specifications this describes what is wanted from the system, not how it is done. It is up to the designer to build in features that prevent the user from getting lost. Whether they do this by adding explicit feedback on the display which describes exactly whereabouts in the menu hierarchy the user is, or by flattening the menu hierarchy somehow to make it harder to get lost, is not an issue for the specification. The specification describes what is wanted in usability terms and how to tell whether or not it is achieved.

Because this scheme is based on normal engineering practice it should fit well into standard engineering practice like the waterfall model. In such an augmented practice as well having to discharge design obligations about the functionality by showing that a solution is correct for its functional specification, the designer also has to discharge the usability specification by showing that it has been solved.

Problems with usability engineering

The problem with this sort of usability specification is that it may miss the point. What really is at issue with the mobile phone example may be not that the user cannot get at all the functionality easily, but whether the functionality that has been crammed it is really useful in the first place. Usability engineering must start from the very beginning of the design process to ensure that account is taken of these issues.

Also, it is not clear that what is specified in the usability specification actually relates to genuine usability. In the example given we decided bailing out was a measure of failure. What if the user is not aware of the possibility that they can bail out? Novice users may struggle on, getting more and more lost and frustrated, unaware that they can just bail out. By the measurement scheme suggested this behaviour would not count as a failure. Furthermore, and rather flippantly, the designer could ensure that no user ever bails out by removing the bail out function altogether. This may sound silly, but designers have been known to do this sort of thing.

Activity 3

Consider the usability specification we have given for the mobile phone menu system. It describes what the problem is, and how we will know that we have solved it, but not the solution. Have a think about the problem and see if you can think up some solutions. We will return to this in a later review question, so make notes and keep hold of them.

A discussion on this activity can be found at the end of the chapter.

Rapid Prototyping

There is a mismatch between designing systems and testing them with users. In the waterfall model a tangible system only appears towards the very end of the process. For most of the process the system exists only as requirements documents, design specifications and ideas in the designers’ heads. The absolute test of a system for usability is to give it to collections of users and see what they do with it. Unfortunately an actual system only exists at the very end of the design process, therefore user testing can only take place at the end of the process and, as discussed earlier, if a big mistake is identified at the end of the process then it is extremely expensive to fix.

Rapid prototyping is a process whereby mock-ups or prototypes of the system are produced all the way through the design process. These prototypes can be then given to users in order to judge their usability. User responses to the prototypes can be judged and feedback passed into the design process to guide the design of the actual system.

Different forms of prototype

Prototypes can be constructed in several ways ranging through levels of interactivity that they offer to the user.

Storyboards are the simplest form of prototype. They are simply rough sketches of the system and its user interface. User are guided through the system by analysts who show the users how they are expected to use the system, and show them what the system is expected to do. The analysts record user responses to the system and feed these back to the designers.

Storyboards may be very simple drawings of the system done on paper, or there are more sophisticated tools available which allow the analysts to create storyboards on a computer using graphic packages and apply some limited animation to the graphics. These allows a little more reality in the storyboards, but in effect the analyst is still in control and steps the users through the system.

In order to give the users more the idea of interaction then limited functionality simulations may be created. These are effectively mock ups of the system, made to look like possible designs of the finished system but with a much restricted set of system functionality behind them.. There are several packages which allow the rapid development of prototypes. They allow the positioning of various interaction objects (buttons, menus, slider bars, etc) on the screen and the attaching of simple behaviours to those objects. Such prototypes give the user the impression of interacting with the full system. The simplicity of the prototypes mean that they can be rapidly reassembled in differing layouts according to user feedback.

Simulations also allow for hardware designs to be cheaply evaluated. Something like a video recorder has built in hardware buttons and controls. Mocking up a physical version may be quite expensive, but a ‘soft’ mock up may be created on a computer, where a picture of the proposed control panel is generated and the user can interact with it using the mouse to ‘push’ the buttons.

Another technique for prototyping is the ‘Wizard of Oz’ technique where the user is given a mock up of the system, which actually contains little or no functionality, but is remotely linked to an analyst who pretends to be the system and makes it respond to the user input appropriately. Such a technique is cheap to set up and very cheap to change, because the analyst just pretends that the system behaves in a different manner. There are ethical issues with misleading subjects in experiments in this way though.

There are also a collection of ‘executable specification’ languages which allow designers to describe software specifications formally in a specification language. These specifications can then be rapidly and automatically converted into working programs, to which interface elements can be attached and a working prototype can be presented to the user. The automatic translation from specification to working program means that the working program will be very rough and inefficient; it requires human software developers to produce really good code, but computers can generate code from a specification that does what it required of it, but in a very inefficient way. The problem with executable specification languages is that they tend to require that the specifications are written in a certain way so that they can be made executable. Non-executable specification languages do not impose such restrictions on developers and are therefore more flexible and powerful tools.

Iterative design

The idea of rapid prototyping is that a mock up of the system is produced, tested, the results of the test are fed back to the developers and the mock up is thrown away. Iterative design pushes the philosophy a little further by relying on mock ups and prototypes, but using those as actual artefacts in the design process instead of throwing them away. An iterative design process collects requirements and then endeavours to produce a tangible, but very crude approximation of the system as quickly as

possible. This approximation can then be user tested and then redesigned accordingly. More detail and refinements can be added to this system and tested as they are added.

Gradually a fully featured, working, and hopefully usable system emerges.

Problems with prototyping

The main problem with prototyping is that no matter how cheaply designers try to make their prototypes and how committed they are to throwing them away, they still have to make design decisions about how to present their prototypes. These decisions can then become ingrained into the final product. There is a factor to consider called ‘design inertia’ which describes how designers, once having made a design decision, are rather reluctant to relinquish that decision, admit it is wrong and change it. This is another reason why good designers put off big decisions as much as possible, knowing that if they make a small mistake early on it is easier and cheaper to rectify than a big early mistake.

Rapid prototyping is about exposing bad design decisions as soon as possible after they have been made. Guidelines and usability engineering are more about trying to get the designer not to make mistakes in the first place. Design inertia is therefore much more prevalent in rapid prototyping development than it is in other user centred design processes. Because prototypes do not get thrown away in iterative design processes then design inertia is even more prevalent.

Furthermore in spotting usability problems by user testing, the designer knows that there is a problem, but not necessarily what causes that problem, or how to fix it. Rapid prototyping identifies symptoms, not illnesses and not cures.

Review Question 5

Rapid prototyping differs in one crucial respect to design by guidelines and usability engineering. What is it? (Hint: refer back to your answer to review question 1.)

Answer to this question can be found at the end of the chapter.

Activity 4

Now lets consider the way that the functionality of the web browser is put on the screen, particularly the buttons along the top of the browser. Look at the space they take up and compare that to how much they actually get used. then think about what is really important about the web. Go back to step one and look at what you wrote about the web: what is important about the web? I wrote that what was really important was the information in the web: it is the web pages themselves that are important not the navigation tools offered by the browser.

So now look again at the space used up by the buttons and how much space that leaves for showing the web pages themselves. Now think about the television analogy: how much space is taken up by the televisions navigation tools (i.e. the channel changing buttons) and how much space is taken up by the screen itself. Look at a typical web browser display. Typically, the navigation buttons occupy about a fifth of the screen, and our analysis shows that only one of those buttons gets any significant use!

Now lets stop being destructive of the web browsers design and start thinking of constructive design ideas. We should have a picture of what functionality the user actually uses and in what situations from activity 3. So using a pen and paper sketch out an interface design which gives prominence and ease of access to the commonly used functions, makes the viewable window with the current web page in it as large as possible and buries unused functions away in menus.

A discussion on this activity can be found at the end of the chapter.

Design Rationale

Design rationale is about explaining why a product has been designed the way it has. Throughout this section we have been describing ways of supporting a designer in making design decisions, i.e.

selecting one design out of the design space. For each decision made there must a set of reasons why that particular decision was made. Design rationale is about recording those decisions and the reasons why they were made.

A design rationale is a useful design tool because it explicitly lays out the reasoning behind a design process and it forces designers to be explicit about what they are doing and why they are doing it. In particular a design rationale can be used after a product has been completed in order to analyse why a product was a success or failure. If a similar product is being designed subsequently then its designers can refer to a design rationale to discover why earlier products were designed the way they were, and with the benefit of hindsight judge whether the earlier design decisions were successful and warrant repeating.

Design rationales are particularly helpful in interactive system design because, as we have been discussing, there is rarely one objectively correct solution to any problem, and some solutions may contradict one another, or require trade-offs. Design rationales require the designer to be explicit about how contradictions were resolved and trade-offs were made.

Furthermore the design space may be very large and therefore it is not obvious that a designer will even consider the best solution, never mind choose it. A design rationale makes it clear which options from the design space were considered and why. If an apparently better solution were later discovered then it is obvious whether that solution had been considered and discarded for some reason, or not considered at all.

Usability is very context dependent; what is good for one user may be dreadful for another. If subsequent designs are made where the context of use does not change then a design rationale can be reused without modification. If however the context does change then new design decisions can be made for this new context, but in the light of the decisions made for the older context.

The QOC (Questions, Options, Criteria) analysis technique is a design rationale. It is a graphical notation that allows designers to visibly lay out the reasoning behind design decisions.

Questions are effectively problems in the terminology we have been using. Options are possible solutions to those problems, and criteria are arguments as to why or why not a given option is appropriate.

Note

Returning to the mobile phone menu problem we discussed in the usability engineering section, a question may be ‘How to prevent users becoming lost in menus?’

Options for this question could be:

1. to flatten menu hierarchies to a maximum depth of three,
2. to give textual feedback as to the current menu display (e.g. display the text ‘Address book : Edit : Add number’ to show that the user has selected the ‘Address book’ option from the main menu, the ‘Edit’ option from the Address book menu, and ‘Add number’ option from the edit menu.
3. to give graphical feedback as to how deep the user is in the menu hierarchy by shifting each new menu selected a little down and to the left, giving the impression of stacking up menus.

For each of these options there are several criteria effecting whether or not it is a good design decision:

1. Screen real estate
2. Limited functionality
3. Accurate user knowledge as to where they are in the hierarchy

Now we can go through discussing each of these criteria for each of the options.

1. If menu depth is kept to a maximum of three then:
   - this has no real effect on screen real estate,
   - it limits the functions that can be placed in the phone. Say a maximum of six items per menu can be displayed, then there is a maximum of 258 functions that the phone can perform,
   - keeping the menu hierarchy flat, improves the chances of the user remembering where they are.
2. If textual feedback is given then:
   - If textual feedback is given then:
   - there is no limit to the number of functions,
   - user knowledge of where they are will be very accurate.
3. If graphical feed back is given then:
   - screen real estate can be efficiently used,
   - there is no limit to the number of functions,
   - the user will have a prompt as to where they are in the hierarchy, but not a very accurate one.

So for the question we have identified three options and three criteria which effect those options.

Now it is up the designers to argue which is the best design option.

Review Question 6

Imagine you are the designer faced with implementing one of the options. Which would you decide to implement and why? Is this a good analysis of the menu design space? Explain your answer.

Answer to this question can be found at the end of the chapter.

Modelling Techniques

In later units we will deal in considerable detail with modeling techniques, but we shall give an overview and introduction here. Generally models are approximations of real world artefacts which can be analysed. A good model is one which fairly accurately predicts the behaviour of the real world artefact it represents while doing away with a lot of the confusing complexity. Models are therefore tools for ‘abstraction’. What is important for an analyst using a modeling technique is that it abstracts away the irrelevancies and keeps the important facts. A model that does so it called ‘well scoped’, it is up to the analyst to chose a modeling techniques that is well scoped for the questions they want to ask of it.

Task analysis

A task analysis technique makes a model of the job that a user is expected to perform. Analysis techniques can be applied to those models in order to determine such facts as how long it may take users to perform given tasks, or how much ‘cognitive load’ is placed on the user (where ‘cognitive load’ is broadly a measure of much information the user needs to remember).

The most researched task analysis technique is GOMS (Goals, Operations, Methods and Selection) developed by Card et al (1983). The analysts describes:

- the user's goals, or the things that user wants to achieve,
- the operations, or the things the user can do, perhaps such things as thinking or looking, or maybe selecting items from computer menus, typing or pointing with the mouse,
- the methods, which are sequences of operations that achieve a goal, and
- selections, which describe how the user may choose between different methods for achieving the same goal.

Once this model of the user's task has been compiled then measurements of the time it takes to perform the operations can be added (these measurements are taken from empirical observations of users) and a prediction of the time it will take the user to perform a task can be calculated.

GOMS only really deals with expert behaviour though and takes no account of the user making errors. It has been argued that a surprisingly large amount of user time is spent making, or recovering from, errors, even for expert users. Therefore the accuracy of GOMS models have been questioned.

GOMS analysts scored a notable victory however when they accurately predicted that users of a new computerised telephone system would take longer to perform their tasks than users using an older, apparently slower system (Gray, John and Atwood. 1993).

User modeling

Whereas task analysis aims to model the jobs that users do, user modelling aims to capture the properties of users. User models can capture and predict properties such as the way the user constructs goals, how users make plans to carry out those goals, the users' ability to do several tasks at once, how the user manages perception, etc. Such models are based to a large extent on psychological theory, which in turn is based on empirical evidence.

Typically the analyst builds a model of the interactive device that is intended to be built and then integrates this device model with an existing user model. This integrated model will be able to predict certain behaviours, and the analyst can therefore gain an idea as to whether the user will be able to reasonable perform the tasks that the analyst wants them to.

Interactive device modeling

Several techniques have taken existing software specifications and analysed them 'from the users' point of view'. In other words the analyst takes a model of the interactive device, which is typically produced as part of the design process, and analyses it for 'usability properties'. A lot of work went into proposing usability properties and then formalising mathematical equivalents of them. In this way software specifications could be mathematically analysed for usability in much the same way as they can be analysed for functional correctness.

Dix's PIE model (Dix 1991) is a classic example of interactive device modelling. The device is modelled as a collection of states with allowable transitions between them. This model can then analysed mathematically for such properties as 'reachability'; the ability of the user to get from such state to any other allowable device state. Dix also formalised what it means for a system to be so called 'WYSIWYG' (what you see is what you get) by separating the displayed output from the printed output in his model and mathematically showing correspondences between the two.

Problems with modeling

As we explained above, models must be used with care. Because they abstract details away the analyst must understand what is being abstracted away and be able to argue why those details are not important. Furthermore modeling is seen as a highly skilled and time consuming activity. The idea of modeling is to be able to predict usability issues before a system is built, but many developers have the impression

that modeling can be so complicated and highly skilled that it is cheaper to just build the system and then test it on users. Refutations to this are rare, but compelling (e.g. the GOMS analysis of the phone system we alluded to above). We will return to these issues in much more depth in subsequent units.

Activity 5

Activity 5 winds up the web browser analysis and redesign process by discussing what we have done. Consider my redesign for the web browser, along with your redesign, and the original design of the browser and compare all three. When there are differences between the three try to argue which is the best' and, most importantly, why it is the best. Use the evidence you gathered in unit 2.

If the original design of the web browser is not good, try and think of why this may be. The designers working at Netscape and Microsoft are not stupid and do not inflict bad designs on their users wilfully, but if something has gone wrong with their design then there must be a reason for this. Hopefully we have developed designs for web browsers that are user centred, if Netscape and Microsoft come up with different designs then this means that they may be working with different priorities we are in this tutorial. What are those priorities, and why do they take precedence over usability. Consider the arguments about feature accretion we discussed in the Contents notes. Do Netscape's and Microsoft's products show signs of feature accretion? If so, why? And do you consider this to be a bad thing? Why?

You will be invited to discuss your analysis in the discussion section.

A discussion on this activity can be found at the end of the chapter.

Evaluation and User Feedback

Landauer states that user evaluation is the ‘gold standard’ for usability. In the previous section we mostly discussed ways that designs can be analysed for usability, largely in the absence of users themselves. Landauer argues that methods for predicting usability are all well and good, but the only real way for evaluating usability is by giving users finished products and analysing how they interact with them.

Evaluating user behaviour is one of the most advanced and well researched fields in HCI.

The classic example of designing with user feedback is the IBM 1984 Olympic Messaging System (See Landauer for a detailed description).

IBM had to quickly develop a messaging system for use by the competitors in the 1984 Olympic games. Because of the huge diversity in users (different languages spoken, different levels of IT competence, different expectations of the system, different cultural backgrounds, etc) there was no way that the designers could accurately predict the usability of the system before the athletes arrived at the Olympic village and started using the system. Furthermore the games only lasted for a few weeks, so there would be no time to correct the system during the games; it had to be right first time. The designers therefore conducted initial user studies with passers-by at the IBM research centre, followed by more extensive trials at a pre-Olympic event with competitors from sixty five countries. The system was then run on a large scale with American users before the opening of the games. Each of these tests identified errors with the system and designers did their best to fix them. The final system that was used at the games was robust and was used extensively without major problems.

Ways of Evaluating User Behaviour

There are many ways of evaluating user behaviour and we shall discuss in more depth in later units. Which techniques should be used depends on what information you need to get from the users. The sort of information you can gather ranges from quantifiable measures of performance time to much more qualitative aspects such as levels of user satisfaction. Which measures are taken and acted upon is dictated largely by the purpose to which the system is to be put. Designers of consumer leisure products are going to be much more interested in levels of user satisfaction than performance measures. However designers of systems where profits rely on performance and whose users are paid to use the

systems are going to be much more interested in performance measures. Gray et al (1993) give an interesting economic figure: they were involved in the redesigning of the workstations used by New England telephone operators. Given the number of operators and the number of calls they took, Gray et al estimate that a one second reduction in work time per call would result in a saving of $3 million per year.

The cost of performing an evaluation can also be a very important factor; performing an evaluation and analysing the results can be very time consuming and costly. Some techniques for evaluation (e.g. questionnaires) are much cheaper than others.

Laboratory set evaluation

This method of evaluation derives from scientific psychological studies. Psychologists attempt to study behaviour experimentally by giving subjects tasks to do in a controlled environment. By controlling the environment psychologists attempt to control variables that may effect the subjects' behaviour. If the experimenter can argue that they held all variables steady except one, and that changing that one variable changes the subjects' behaviour, then the experimenter is in a good position to argue that the one variable has a causal effect on the subject's behaviour.

Laboratory set evaluations are intended to produce results that have a high level of scientific rigour to them. Typically users will be invited to perform certain tasks using a system under laboratory conditions, then other users may be invited to perform the same tasks using a variation on the system. If the two sets of users behave in different ways then the experimenter can claim that the difference in behaviour are caused by the differences in the system.

Users' behaviour can be recorded by videotaping or by programming the system to automatically record what the users do in log files. After the experiment the analysts will usually try to get more qualitative responses to the system from users by asking them to fill in questionnaires or by interviewing them.

The main criticism levelled at laboratory set evaluations is that they lack 'real world validity'. Users may behave in a certain way in the rather unusual setting of a laboratory, but there is no guarantee that they will not behave in a completely different way in the 'real world'. Furthermore care needs to be taken with the users that are asked to perform experiments. Many psychological experiments that are reported widely are undertaken in universities by researchers who use the undergraduate population as subjects. On deeper investigation you will find that psychological statements about behaviour do not really apply to the population as a whole, only to a rather small and demographically strange set of university students. Evaluators of interactive systems should take similar care to evaluate their systems with the sort of people who are actually going to be using it.

Laboratory set evaluation is a way achieving scientific rigour in an evaluation, which is a very strong requirement for an evaluator to set themselves. Unless the evaluator wants to publish their results in learned journals there are easier and cheaper ways of getting data about user behaviour. Furthermore laboratory investigations require skilled evaluators and specialised equipment in the laboratory; they can be very expensive.

Ethnographic studies

An ethnographic study is a way of getting round the problems of real world validity presented by laboratory set evaluations. Ethnographic studies realise that valid user behaviour is not to be found in laboratories but in the users' homes and workplaces. Ethnographers also realise that users behaviour is also influenced by the presence of the experimenter. Hence the experimenter will try and become part of the users' environment. If a study is being made of a system in a workplace then the experimenter will join the workforce and perform the tasks that the users do.

Because the experimenter cannot control the environment to anywhere near the same extent as can be done in laboratories then it is much more difficult to make claims of cause and effect. Ethnographic studies are a newly emerging way of studying behaviour and are beginning to gain respect for the real world and valuable insights they give into behaviour. New sets of procedures for performing

ethnographic studies are emerging and as a field ethnography is rapidly moving towards scientific respectability.

Much care must be taken with ethnographic studies though; they must be studies of the users' behaviour and not of the experimenter's. Because the barriers between users and experimenters are deliberately broken down there must be good evidence in the evaluation that the experimenter is reporting the users' behaviour and not their own. Although an explicit laboratory is not required, ethnography is still expensive in terms of experimenter skill and time.

Questionnaires

Questionnaires are a cheap way of gathering opinions from a large number of users. They range in how prescriptive they are in what sort of answers the user can give. They can ask open questions and leave the user space to respond by writing free text, or they can give very specific questions with a set range of answers that can be ticked. The answers on questionnaires can be read automatically if they are 'tick box' answers as opposed to free text. Being able to read answers automatically can also dramatically decrease the costs involved in the evaluation. Web pages can also be written in the form of questionnaires so that users can automatically send information to the developers.

There is a trade off between the amount of freedom given to the user in how they fill questionnaires and the time and effort required to analyse the questionnaires. A questionnaire made up entirely of questions with a set of answers that the user must tick allow very little freedom, but are very quick and easy to analyse. The more you allow users to fill in free text, the more effort is involved in analysing them.

The questions need to be carefully written so that users can understand them and give useful answers. Users will soon get bored and fill in fallacious answers if they do not understand the questions or find them irrelevant. Furthermore questionnaires should be fairly short and to the point to prevent users getting bored. If a lot of questions are to be included then it is best to put the important ones first so that users answer them before giving up. Because filling in questionnaires is such a boring task for most users evaluators will often offer incentives to users to complete the questionnaires.

User interviews

Interviewing users is quite skilled; the interviewer needs to be able to get appropriate information out of the users, while leaving the interview open enough to let users add their own opinions, while not letting the users 'run away' with the interview and discuss things of interest to them, but not much to do with the system being evaluated.

Summary

Each of these approaches to evaluation have their own weaknesses and strengths. A really extensive evaluation will make use of many, if not all of these techniques in order to try and maximise the benefits of each. A small evaluation will probably get best results by conducting several user interviews; they tend to generate the best quality information. Although evaluating by questionnaire is appealing because of its cheapness, it should be applied with care; a badly designed questionnaire can generate misleading results.

Review Question 7

Each of these approaches to evaluation have their own weaknesses and strengths. A really extensive evaluation will make use of many, if not all of these techniques in order to try and maximise the benefits of each. A small evaluation will probably get best results by conducting several user interviews; they tend to generate the best quality information. Although evaluating by questionnaire is appealing because of its cheapness, it should be applied with care; a badly designed questionnaire can generate misleading results.

Answer to this question can be found at the end of the chapter.

Problems with Evaluation and User Feedback

The main problem with user evaluation in designing interactive systems is that information about what, in usability terms, is wrong with a system comes very late in the design process. Recall that we argued that rectifying mistakes becomes more and more expensive the later in the design process they are identified. This means that usability issues identified by evaluation can be very expensive to remedy. Indeed in a lot of cases, too expensive to remedy and the product will get shipped full of usability bugs.

There is plenty of evidence of developers identifying problems by user testing, realising that it is now too expensive to fix them, papering over the cracks and shipping the product. A very popular way of papering over cracks is by creatively writing the user manual for the product. If a usability problem is identified then the developers can usually think of a way of getting around the problem, though this get around is usually complicated and difficult. The problem and the get around can be described in the manual, and to a rather trivial extent, this ‘solves’ the problem. Usability expert Harold Thimbleby claims that the usability of a product is inversely proportional to the size of its user manual; the bigger the manual, the less usable the product.

Many of the criticisms that can be aimed at rapid prototyping can also be aimed at evaluation techniques. In particular evaluation techniques identify where a system is unusable, but not why.

Review Question 8

Why is identifying a problem and a get around' solution and then describing that in the user manual not a sufficient way of solving' usability problems.

Answer to this question can be found at the end of the chapter.

Guidelines for User Centred Designing

In this unit we have given a broad outline of the techniques available that can be used to improve the usability of an interactive system. The question is: which technique is best? The answer is: it depends. The most important thing to realise is that there is no usability panacea. To be successful a developer must understand what techniques are good at and what they are not good at.

For example task analyses are good an analysing systems where the user has a set, explicit goal that they will try to achieve. They are not so good at analysing systems where the user acts in a more exploratory way. Task analysis would therefore not be very good at analysing web surfing systems, but would be much better at analysing an accountancy package.

The ultimate test of usability for a system is to analyse how users behave with it. What we will promote in this course is the idea that developers can design with users in mind, and therefore improve usability before getting to the evaluation stage. We are, however, not advocating designing interactive systems in isolation from users.

One of the main themes of this unit has been that making changes to system designs is fairly cheap and painless, whereas making changes to finished products is very expensive. Ideally we would recommend a design process where usability and usefulness are considered throughout the process. Therefore a system should be arrived at which is less likely to suffer from serious, large scale usability problems. Smaller scale interface issues can then be identified and dealt with by evaluation techniques.

There is no ‘right’ way of designing a system to be free of usability errors. The benefit of most of the techniques we have outlined in this unit are not that they tell a designer how to improve usability, but that they force the designer to think about the system from a ‘usability point of view’.

Below is a summary of the user centred design techniques we have discussed in the unit, stating in simple terms what they are good at and not good at. Once you have completed this course you should be more familiar with many of these techniques and should therefore be able to return to this table and add to and discuss it in more detail.

Technique	Good For	Bad For
Guidelines	Actually telling designers what they should or should not to achieve those goals.	Actually telling designers what they should or should not to achieve those goals.
Usability engineering	Actually telling designers what they should or should not to achieve those goals.	Actually telling designers what they should or should not to achieve those goals.
Rapid prototyping	Quickly testing designs with a user population.	Good software engineering – rapid prototyping can force design decisions to be made too early in the design process.
Design rationale	Recording why a system is designed the way it is.	Allowing developers to think that all designers options have been addressed.
Modelling techniques	Stripping away irrelevancies in a design so that the essentials can be easily analysed.	Allowing analysts to strip away essentials and analyse irrelevancies.
Evaluation techniques	Getting actual hard facts about what users do or think about systems.	Spotting big mistakes early enough that they can be cheaply remedied.

Extensions

Other development cycles

Obtain a good software engineering textbook (e.g. ‘Software Engineering’ by S. R. Schach, 1993, published by Irwin and Associates. A search for ‘software engineering’ on any technical library catalogue should bring up a wealth of good text books.) and investigate some of the design processes other than the waterfall model. Discuss the implications of user centred design for those process. Do those processes make user centred design easier or not? Why?

In particular you might wish to investigate the ‘Delta model’ which is held to be particularly adept at incorporating user centred design. What makes this so?

Is usability a science or craft skill?

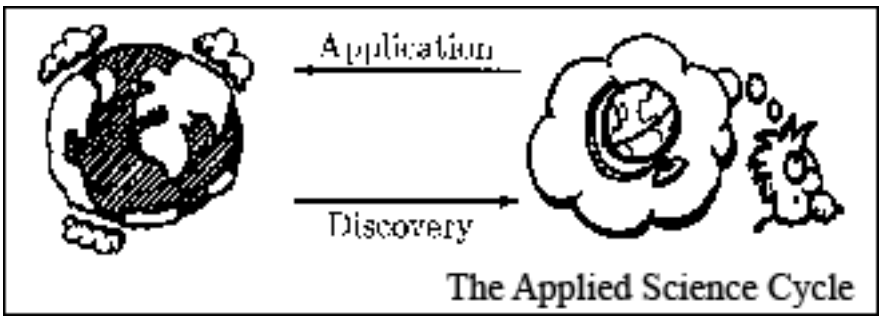
Newton predicted that if you dropped a hammer and a feather in a vacuum they would fall at the same rate. This is because gravity pulls on objects consistently, independent of their mass. The only reason that hammers and feathers fall at different rates is because of friction with the air. Remove the air and they fall at the same rate. Newton was able to make this prediction based on a thorough and correct (not withstanding Einstein and relativity) understanding of the science of gravity. It was not until some hundreds of year later that an Apollo astronaut actually took a feather and hammer and dropped them on the moon that we actually had evidence that Newton was correct.

There is an equivalent question for HCI; do we understand users and user behaviour enough that we can predict how they will behave with an interactive system? If we do, then we can design for usability because we can predict user behaviour, but if we cannot then the best we can do is build a system, give it to users and see what they do with it.

The applied science cycle

Barnard (1991) suggests the ‘Applied Science Cycle’ shown in figure 1. There is a ‘real world’ of tangible objects and observable behaviours and an ‘abstract world’ of scientific theories about the sort

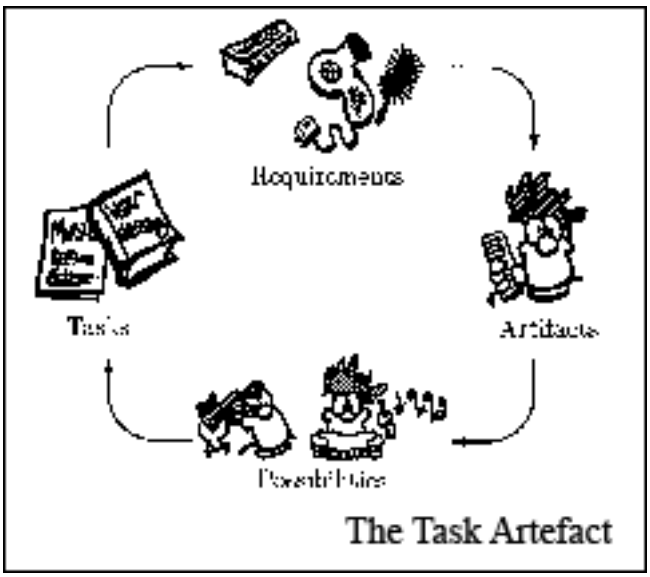
of things that go on in the real world. A scientific theory is held to be a good if it manages to accurately predict what goes on in the real world. (Newton's theories of gravity are good, even though they have been supplanted by Einstein's theories. Newton is accurate when describing gravity as experienced by those who are not approaching the speed of light. As most of us rarely approach the speed of light then Newton's theories are good for us.)



Barnard argues that things go on in the real world and that scientists observe them and think up theories to account for them. This translation from the real world to the abstract is called 'discovery'. Designers can use the theories in the abstract world to design artefacts in the real world. This translation from the abstract world to the real world is called 'application'.

Applying Barnard's cycle to HCI, we have a world view whereby people who study human behaviour (psychologists and sociologists) observe and analyse human behaviour and develop theories about their behaviour (discovery). Interactive system designers can then apply these theories to help them better design their systems (application).

The task artefact cycle



Carroll et al (1991) rejects Barnard's cycle on the grounds that is inappropriate for designing interactive systems. He argues that human behaviour is simply too complex to usefully abstract over and therefore the discovery translation of the applied science cycle is worthless.

Carroll suggests a replacement for the applied science cycle called the 'Task-artefact cycle' shown in figure 2. It is rather more subtle than Barnard's cycle so we have shown it with an example.

We start with artefacts; tools for doing a job. Here the tool we consider is a comb. A tool offers possibilities, and those possibilities may not be the ones for which it was originally designed. In the case of the comb it may be used for combing hair, or it may be wrapped in paper and played as a

musical instrument. This gives us new tasks. From these tasks we can gather requirements from users about how they may wish to do their tasks better, in the case of the comb the user may require a harmonica if they wish to concentrate on their musical skills or better hair dressing tools if they stick to the comb's original purpose.

Central to Carroll's cycle is the notion of artefacts, whereas central to Barnard's cycle is abstract theories. Carroll therefore argues that his cycle is more practical and better reflects design practice in the real world. However Carroll's argument is that you must produce something and set it in the real world before you will be able to usefully understand how it is used. Taken to its logical conclusion Carroll's argument is for throwing mud against walls.

Barnard's argument should not really interpreted as a description of how things are, but how they should be. Useful and accurate theories of human computer interaction mean that systems can be designed based on those theories, by anyone who understands them.

The counter argument to theory based HCI is that of 'craft skill'. A craft skill is something that cannot be caught in a theory. Most artistic endeavours rely on craft skill. (Actually most scientists think that artistic endeavours rely on craft skill – when analysed most artists have a plethora of theories and rules which determine what they do.) Many HCI practitioners work as 'craft skill' experts; they will inform designers about what makes a good or bad interactive system, but the rationale behind their arguments is unclear and ill-formed. They have knowledge about HCI, but not in a form that is expressible. Hence they do not advance HCI as a science.

Carroll's arguments, particularly the argument that human behaviour cannot be abstracted over, lean towards HCI as a craft skill. It is not clear how a designer goes from 'requirements' in his cycle to 'artefacts' other than by applying craft skill.

Psychology and sociology as useful theories for HCI?

There is a well researched theory known as Fitt's Law (1954), developed by a psychologist. It describes very accurately the time it takes for the hand to move a certain distance and hit a target area. This law is very useful for designing buttons in graphical user interfaces (see Dix et al, page 252). It allows the designer to analyse the size and spacing of buttons and predict how easily the user (with the mouse) can press them.

Fitt's Law is a classic example of a discovery on Barnard's cycle which can be usefully applied. There are other examples of psychological theory which can be used to predict user behaviour. May, Scott and Barnard (1996) have developed theories of perception which predict how well users will be able to pick out icons on screen.

These 'small granularity' theories abound in psychology, describing hand to eye co-ordination, perception, our ability to think about two things at once, our ability to reason logically, etc.

Sociology looks at human behaviour at a much larger 'granularity', describing human group behaviour, particularly of interest to HCI is 'work theory' which describes how we go about doing our jobs and fulfilling tasks.

HCI sits between psychology and sociology. Interactive systems are seen as tools in work, so HCI is smaller scale than most sociology theories, but larger than most theories propounded in psychology. There are theories of HCI but they are not as well researched and established as those drawn from psychology and sociology. Also they are not clearly or abstractly set out as theories (at least not as abstractly set out as Fitt's Law, which is one mathematical equation) or as general as could be hoped.

Summary

In this extension unit we have dealt with some fairly lofty ideas about scientific validity and the use of science in design. The aim of the extension is to argue that we can design for usability, rather than

just test for usability. We believe that it is possible and desirable to design for usability, although this approach is contentious.

Answers and Discussions

Answer to Review Question 1

Design techniques are applied during the design process in order to try and produce a system that is more usable. Summative evaluation techniques are applied when most of the design has been done and a tangible system has been produced. Evaluation is performed by giving a system to users and studying how they use it. Therefore evaluation can only take place when there is a tangible system to study. While design is taking place there will not be such a tangible system.

Answer to Review Question 2

During the early stages of the waterfall design process nothing tangible is actually produced. The early phases are about having and organising ideas about how to go about solving problems. These ideas exist on bits of paper or completely abstractly in the designers' heads. They are therefore quite cheap to change if a mistake is found.

Later in the process actual tangible products are generated. These are much more expensive to change.

Answer to Review Question 3

A correct design step selects an option from the design space that solves the required problem. A good design step selects an option from the design space that solves the required problem well (e.g. efficiently, cheaply, etc). A good design step is by necessity correct, but not necessarily vice versa.

Answer to Review Question 4

Evolutionary development is best described by the task artefact cycle. It describes how the use of objects can change through time, and how the design of those objects needs to be updated to match those changed uses. This change of needs, use and objects through time is what evolutionary development is all about.

Answer to Review Question 5

Rapid prototyping actually involves users in the design, whereas the other techniques for user centred design rely on more abstract theories of user behaviour.

Answer to Review Question 6

Based on the analysis given the first option is probably the best. Screen real estate is the largest consideration to be made and therefore the second option is ruled out on those grounds. The third option is appealing, but even so it would still possible for the user to get lost in a deep menu hierarchy. The first option may limit the telephone functionality, but 258 functions should be far more than sufficient for a mobile phone. If more than 258 functions are required, this should raise much more fundamental questions about the design of the phone. The fact that the second option gives the user perfect knowledge as to where they are can be discounted to a great extent, because giving the user perfect knowledge is probably unnecessary, and that giving the user hints is probably sufficient.

However the option space we have described in the example is small. In particular there is no reason why the designer cannot implement combinations of the options. In this light a combination of options one and three gives a very good solution as it would have no seriously negative criteria.

Answer to Review Question 7

Because of the size of the user population devising a questionnaire would be the best solution. As well as designing the questionnaire well you need to consider how you are going to induce users to answer it. If you are delivering the product over the web then you could make answering the questionnaire a preliminary to downloading the product (but beware, this means the user will not have actually used the product). You may instead wish to offer an incentive like free support if the user completes the questionnaire.

Answer to Review Question 8

Mainly because users do not read manuals. This is not the users' fault; they have much better things to do with their time than trawl through manuals, none of which make very interesting reading. A well designed, usable consumer product should not need a manual.

Discussion point

Discuss the process you went through in this unit and unit 2 to analyse and redesign a web browser. What have you learnt? In particular we wanted to show that designing for usability need not be very difficult. In subsequent units you will be introduced to some complicated techniques that may daunt you.

Consider the amount of effort you have put into this analysis: it should have taken you about three hours. But with just a few hours of gathering evidence and careful critical thinking we have come up with a redesigned web browser that we claim is better suited to what we do with the web. We have employed no special skills other than critical thinking. Microsoft spend a huge amount of money per year on usability testing, but we have suggested improvements to one of their most visible products in a few, cheap hours. Why is that? Who is in the wrong? Us or Microsoft? Why do Microsoft spend this amount of money when it apparently does not greatly improve their products?

Discussion of Activity 1

Returning to the mobile phone menu problem we discussed in the usability engineering section, a question may be 'How to prevent users becoming lost in menus?'

Options for this question could be:

1. to flatten menu hierarchies to a maximum depth of three,

2. to give textual feedback as to the current menu display (e.g. display the text 'Address book : Edit : Add number' to show that the user has selected the 'Address book' option from the main menu, the 'Edit' option from the Address book menu, and 'Add number' option from the edit menu.

3. to give graphical feedback as to how deep the user is in the menu hierarchy by shifting each new menu selected a little down and to the left, giving the impression of stacking up menus.

For each of these options there are several criteria effecting whether or not it is a good design decision:

1. Screen real estate

2. Limited functionality

3. Accurate user knowledge as to where they are in the hierarchy

Now we can go through discussing each of these criteria for each of the options.

1. If menu depth is kept to a maximum of three then:

   - this has no real effect on screen real estate,

- it limits the functions that can be placed in the phone. Say a maximum of six items per menu can be displayed, then there is a maximum of 258 functions that the phone can perform,
- keeping the menu hierarchy flat, improves the chances of the user remembering where they are.

2. If textual feedback is given then:
   - there will be problems with screen real estate, textual descriptions take up a lot of space,
   - there is no limit to the number of functions,
   - user knowledge of where they are will be very accurate.
3. If graphical feed back is given then:
   - screen real estate can be efficiently used,
   - there is no limit to the number of functions,
   - the user will have a prompt as to where they are in the hierarchy, but not a very accurate one.

So for the question we have identified three options and three criteria which effect those options.

Discussion of Activity 2

The answer is to where the browser finishes up after press Back twice is: ‘it depends’. If you moved from page to D by following links then pressing the back button twice will put you on page C. If you went from D to B by pressing the Back button twice then pressing the Back button twice on page H will send you to page A.

It turns out that what the Back button does is simple: it moves you back one place in the list of pages on the Go menu, but what is actually on the Go menu is not easy to work out, and therefore what the Back actually does is not very predictable. If you are on the top page on the Go menu then jumping along a hyperlink adds the new page you jump to on to the top of the Go menu. If, however, you jump along a hyperlink when not at the top of the Go menu then the pages above the current one are deleted from the Go menu and replaced by the page jumped to. The Go menu is therefore no a complete history of where the browser has been. Many users are not aware of this.

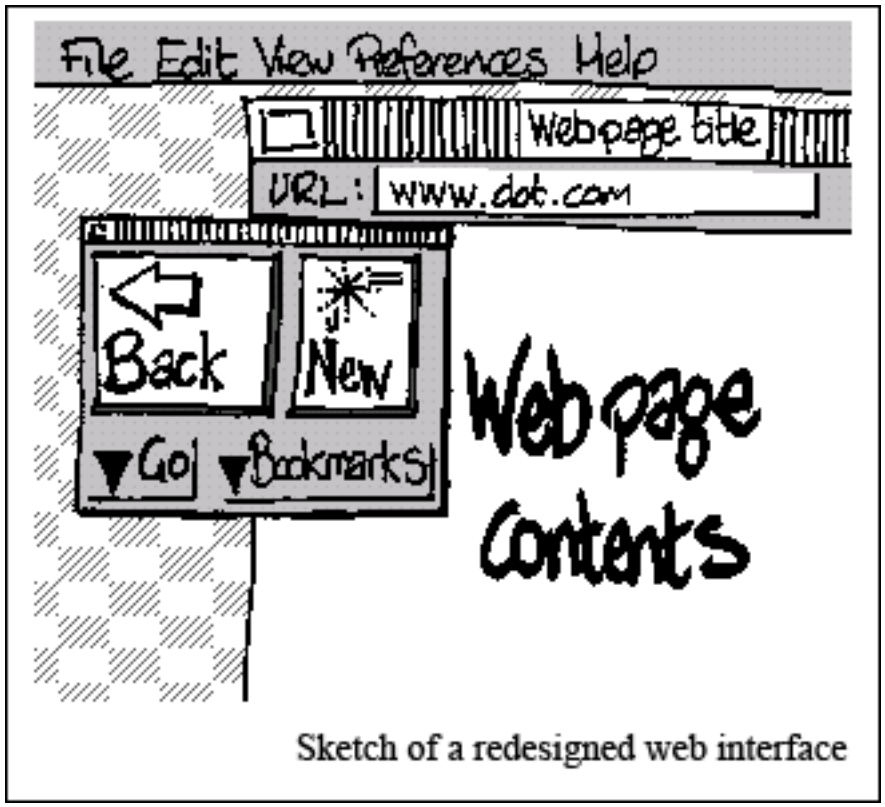
Discussion of Activity 3

Think about a menu system on a normal computer with a full sized screen. It is difficult for the user to get lost within that menu system, because the computer gives very explicit feedback as to where the user is in the menu hierarchy. On a restricted size screen there is no room to give the user such obvious feedback as to where they are, so you must consider (at least) two things: ways of giving the user feedback as to where they are which uses very little screen real estate, and ways of making the menu hierarchy simple enough that the user does not get lost.

Discussion of Activity 4

A sketch of my interface is shown below.I have made the navigation controls into a floating window and made the back button largest in that window. The Go menu now drops down from the navigation window and the bookmarks can also be shown in a floating window. The URL of the current page is shown across the top of the web page window and can be edited. All other functionality is in the menus. I have moved the ‘New navigator’ function out of the menus and onto the navigation window. I use multiple windows a lot and find it annoying having to rattle about in the menus to create a new browser. Also, instead of starting a new browser with the predefined ‘home page’ it creates a copy of the current page. This is so that when I find a page with several links on it that are interesting and I

may want to come back to I can easily make a copy of that page and come back to it later to investigate the other links.



Sketch of a redesigned web interface

I have sketched out this interface to support my use of the web, your use may be different, and therefore you will have hopefully come up with a different design. So much the better.

Discussion of Activity 5

If Mozilla Firefox and Microsoft Internet Explorer show feature accretion then that is because Mozilla and Microsoft consider that loading their applications up with as many features as possible is an advantage. Putting many features in a product can give that product the appearance of being ‘powerful’ or ‘professional’. An argument that those features are useless, and worse, get in the way, is much less tangible and difficult to demonstrate, and therefore does not become much of a negative point.