Human Computer Interaction

UNIVERSITY OF CAMBRIDGE

Computer Laboratory
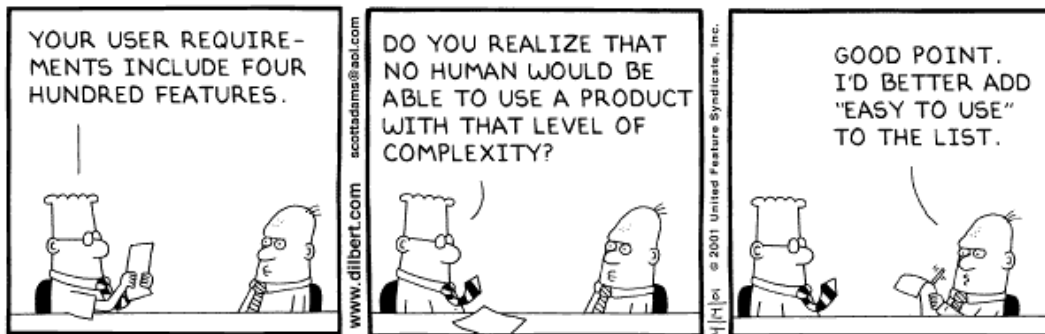
Part II

Alan Blackwell, 2006–2007

http://www.cl.cam. ac.uk/Teaching/current/HCI/

Computer Laboratory
William Gates Building
J J Thomson Avenue
Cambridge CB3 0FD

E-mail: afb21@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~afb21/

Human Computer Interaction Notes

Dr Alan Blackwell
University of Cambridge Computer Laboratory

Part II course 2006-2007

0.1. Introduction

With the exception of some embedded software and operating system code, the success of a software product is determined by the humans who use the product. This course deals with theoretical and practical techniques for making successful and usable software. Many computer science students enjoy creating attractive "sexy" graphical displays for their software, but that is not our main concern here. People's idea of what is attractive depends largely on fashion. The software of ten years ago was considered attractive at the time, as were the cars of twenty years ago. The sexiest software of today will look just as old-fashioned before long. Instead, we are interested in the principles that make software more usable for its purpose. Software that is usable for its purpose is sometimes described by programmers as "intuitive" (easy to learn, easy to remember, easy to apply to new problems) or "powerful" (efficient, effective). These terms are vague and unscientific, but they point in the right direction. This first section of the course presents scientific approaches to making software that is "intuitive" and "powerful".

- -
Lecture 1 starts about here -
- -

The field of human-computer interaction (HCI) is rather different from other areas of computer science. In order to make software that is easy to learn, easy to remember and easy to apply to new problems, we must understand something about learning, memory and problem solving. These are topics that are normally taught in cognitive psychology departments rather than computer science (although computer scientists who work in artificial intelligence are very familiar with relevant psychological principles). The other part of the field – how to make products that are efficient and effective – is the central concern of engineering, and is more often taught in engineering departments. Human-computer interaction is therefore partly psychology (plus other social sciences) and partly engineering (plus other professional skills). Some computer scientists find that these new skills and perspectives very difficult compared to what they are used to. HCI research is mostly about experimental studies and design techniques rather than mathematics or algorithms, so serious study of HCI involves more than just new subject matter - it involves new ways of studying. This includes the possibility that there may be alternative theories, each of which provide useful perspectives in different circumstances, but none of which can be proven to be the "true" theory of human behaviour, and more than Java, ML, Perl or BCPL can be proven to be the "true" and perfect programming language.

HCI helps us to understand why some software products are good and other software is bad. But sadly it is not a guaranteed formula for creating a successful product. In this sense it is like architecture or product design. Architects and product designers need a thorough technical grasp of the materials they work with, but the success of their work depends on the creative application of this technical knowledge. This creativity is a craft skill that is normally learned by working with a master designer in a studio, or from case studies of successful designs. A computer science course does not provide sufficient time for this kind of training in creative design, but it can provide the essential elements: an understanding of the user's needs (from psychology and social science), and an understanding of potential solutions (from engineering, plus some architecture and product design). Along the way we will meet some case study

material, and more is available online (see the reference to the Interface Hall of Shame), but much of the content of this course will only come alive once you have graduated and are working alongside experienced software designers.

0.2. Overview

This course includes both theoretical material and practical approaches to designing user interfaces. In the lectures, the two will be combined. There are four general themes, each one of which will be presented together with research background, relevant theoretical material, and then the description of a practical design approach that has been developed from that material. In the notes, all of the theoretical material is organised together for continuity, and the four design methods will be described in more detail toward the end of the notes. I have given a rough indication of where the breaks between these topics are likely to occur in terms of lecture-hours, but the exact position will of course vary.

The theoretical topics include:
- Styles of interaction, and how these have developed from the interface hardware that is available.
- Direct manipulation as an approach to graphical interfaces.
- Human "input" and "output" and how well these match the output and input of the computer.
- The "Model Human Processor" as a way of predicting human performance.
- Mental models of devices, and their relevance to human problem-solving.
- Experimental and prototyping techniques
- Specification and design of user interfaces
- Current research topics in HCI

The four design methods are:
- Heuristic Evaluation (included in Theme 1)
- Keystroke-Level Modeling (included in Theme 2)
- Cognitive Walkthrough (included in Theme 3)
- Cognitive Dimensions of Notations (included in Theme 4)

At the end of the notes, some further resources are given – reviews of the set texts, some other useful books, and a list of online resources. There are also a few exercises designed to provide some insight into how practical usability evaluation might be attempted in a commercial context. It is important to attempt at least one of these exercises before starting supervisions for this course – previous experience with this material suggests that there is little benefit in supervisions unless a range of these exercises have been attempted by the supervisees.

Theme 1. Interaction Techniques

1.1. Styles of interaction in the development of user interfaces

Technical descriptions of human-computer interaction have always been heavily influenced by the physical interaction devices that are available. The earliest digital computers were scientific instruments. Interacting with them involved configuration of the equipment (often mechanical reconnection via wiring panels), setting parameters on a control panel (using switches and dials) and monitoring of processes (via lamps and cathode ray tubes). Improvements in usability would be made in the same way as for the usability of any other machine - arranging the control panel more conveniently, or providing more switches for configuration. The study of usability of machine controls is usually called "human factors" or "ergonomics". It is a separate academic field from HCI, although human factors conferences have recently taken more interest in HCI problems (including Ergonomics 2006 in Cambridge). Some HCI researchers have described their work as "cognitive ergonomics", in order to make it clear that they are most interested in the way that the user thinks about the system.

1.1.1. Algebraic Languages

Once programs started to be designed and written at a desk and loaded into the computer from paper tapes by technicians, interaction with the machine became more like a mathematical activity. The first programming language, FORTRAN, was described as a "formula translator" that automatically interpreted the natural mathematical terminology of these users.

1.1.2. Data Files

Commercial data-processing tasks used an interface that was already established for data storage and processing: punched cards. Keypunch operators created data records using card-punch machines. Punch cards were also used to store programs with one line on each card. Programmers used to walk around carrying file-boxes of cards, and interaction became more like a paper record-keeping operation.

1.1.3. Command Lines

Computers could interact directly with an operator via a teletype terminal. These were used first by the operators of large computers, and then for interacting directly with the first generation of "time-sharing" mainframes and mini-computers. As you typed, the characters were printed so that you could see your input. The computer (or person) at the other end of a teletype link has to wait for the end of your input line, then send a line in response to be printed at your end. This alternation encouraged the view of the computer as another person on a communicating teletype, and interaction as a *dialogue* with a conversational partner. The user issues a command, and the response from the partner (or perhaps subordinate) is an acknowledgement of the command.

1.1.4. Line Editors

This was the style of interaction in use when users started to edit text on-line, rather than by inserting or replacing punch cards in a card file. When using a command-line editor, the user issued commands (paraphrased here) such as:
- Show me what is in line 12 (system responds "The quick brown foz")
- Replace "z" with "x" (system responds "OK>")
- Show me what is in line 13 (system responds "gggggg")

- Delete line 13 (system responds "OK>")

Of course the actual commands were more like "P12 S/Z/X D13". The ideal of human computer interaction was to make this dialogue seem as much as possible like dialogue with another person - ideally by making the commands natural English sentences - while not requiring too much typing. Command languages (see the UNIX command line for an example) were therefore optimised to be brief, but easy to remember and structured like English with verbs and objects.

1.1.5. What You See Is What You Get

Teletypes used lots of paper to print the progress of a conversation, and this was not really necessary when so many commands were only of transient interest. The **glass teletype** or video display terminal displayed the operator commands and teletype responses on the screen, (sc)rolling up the screen like the paper unrolling from a teletype. This was OK for interaction dialogue, but advanced devices allowed control codes that would write characters anywhere on the screen. The next innovation was a "full-screen" editor that showed a whole lot of text at once, rather than one line at a time. They incorporated the idea that the user really needs to see the product they are working on, rather than a transient sequence of commands and responses - What You See Is What You Get or **WYSIWYG**.

1.1.6. Modeless Interaction

The first full screen editors were like front-end viewers added onto existing line editors (the vi editor under UNIX has been a particularly successful example of this, and its command style still exists as a "living fossil" of those days). Many of the commands were issued from a command line at the bottom of the screen, and the user had to toggle between command mode and edit mode. The user interface behaved quite differently in each mode, and it was difficult to remember which mode you were in – often with unpredictable results. Some of the earliest modern-style user studies were conducted by Larry Tesler, later the manager of the Apple development team that developed the original interface ideas for the Macintosh, who found just how difficult "modal" vi-style commands were by observing the way that they caused users constantly to make errors. More advanced editors such as EMACS attempted to provide a **modeless interface**, in which a given keystroke would have the same effect in any context within the interface.

1.1.7. Menus

When users forgot a command, many command line interpreters could send a set of choices to the teletype, so that the user could enter the code for the one he or she wanted. Video terminals with cursor addressing meant that instead of having to remember some huge number of possible commands in order to enter a command-line code, the user could move the cursor to the place on the screen where the desired command was displayed, and press a function key to act on it. This development applied the psychological principle that people find it far easier to **recognise** something they have learned than to **recall** it without any prompt. This was a breakthrough that also allowed the use of longer, natural language commands, rather than short commands to optimize typing time (another living fossil, still found in the 2-letter command namess of unix). At the same time, interaction became far less like a dialogue, because the user no longer typed commands as "sentences", and the system did not provide a text response to every command (observing the effect of a menu command is often sufficient **feedback** for a user to confirm that the system has responded correctly).

1.1.8. Pointing devices

Moving the cursor to the appropriate position in a menu was facilitated by the use of devices such as the *light pen* for pointing at locations on the screen. The light pen was originally developed for graphical applications such as Sutherland's Sketchpad (see a description of this system in a CL technical report). A long programme of human factors experiments led to the development of the *mouse* for specifying relative motion along two axes rather than absolute position – this was a valuable innovation that made it easier for users to point directly at locations within text being typed.

1.1.9. Graphical displays

Video display terminals of the "glass teletype" variety were only capable of displaying 25 rows of 80 alphanumeric characters, but some technical applications required the display of graphical information. Graphical terminals could interpret a variety of complex languages that specified the points and lines to be displayed, forming engineering drawings or electronic schematics. Graphic terminals often switched between teletype mode and graphic mode, rather than drawing command text anywhere on the screen.

1.1.10. Icons and windows

The development of the *bit-mapped display* in personal computers allowed the display of more realistic pictorial images. When combined with a mouse, programs could be controlled by pointing at images. The symbolic content of nodes in schematic diagrams could now represent things that were originally abstract components of a formulaic sentence. The idea of making pictures of abstract concepts, was described as *icons*.

1.1.11. Summary

This brief history demonstrates that the main theoretical approaches to user interface - algebraic languages, dialogues, WYSIWYG, icons - all depended primarily on developments in I/O devices. Some theories of HCI now appear less important than they once did, because the specific interaction problems they addressed no longer annoy us every day. However the resulting design principles are still essentials of user interaction, and have been encoded in the form of design guidelines.

It is likely that future developments in the technology of user interface will be just as radical. When we are all interacting with our wristwatches by speaking to them, studies of accuracy in pointing will seem far less important. Nevertheless, this deepening historical heritage gives us a rich perspective for analysing the way that we use computers.

Lecture 2 starts about here -

1.2. Direct Manipulation

Our current generation of HCI theory is based on analysis of the current generation of technology: *windows*, *icons*, *menus* and *pointers* (the *WIMP* interface). This generation does represent some very significant advances over command line and teletype interfaces. The most radical of these is that the "command" is no longer the central unit of interaction. Instead, the object of the user action (represented by an icon) is the central unit of interaction. This is derived from early computer graphics systems (notably Sutherland's Sketchpad in 1963), in which a light pen could be used to point at and manipulate parts of the drawing. Researchers at Stanford and the Xerox Palo Alto Research Center realised in the 1970s that graphical objects

could be used as iconic representations of abstract data, and that manipulating the graphical object might correspond to commands on that data. This approach was adopted by first Apple Computer then Microsoft in the 1980s. A psychologist, Ben Shneiderman, expressed the important attributes of *direct manipulation* in 1983:

- An object that is of interest to the user should be continuously *visible* in the form of a graphical representation on the screen

- Operations on objects should involve physical *actions* (using a pointing device to manipulate the graphical representation) instead of commands with complex syntax

- The actions that the user makes should be *rapid*, should offer *incremental* changes over the previous situation, and should be *reversible*

- The *effect of actions* should immediately be visible, so that the user knows what has happened

- There should be a modest set of commands doing everything that a novice might need, but it should be possible to expand these, gaining access to more functions as the user develops expertise.

It may seem surprising that these desirable attributes were not always recognised. They certainly seem obvious now that we are familiar with graphical user interfaces, but they are clearly not true of interaction techniques such as the UNIX or DOS command line. You may like to compare your favourite command line to a graphical desktop, and see how it compares on each of these criteria.

1.3. Style Guidelines

Competing operating system vendors are now very concerned that their systems should appear easy to use. Unfortunately, users' experience of usability is often through programs from third party software companies which the operating system vendor does not have control over. In order to make this software easier to use, operating system suppliers, starting with Apple, therefore publish *style guidelines* advising developers on how to make usable applications. The current version of the guidelines published by Microsoft gives some general advice on usability that is almost exactly the same as that published by Shneiderman in 1983 (although with slight differences in terminology including "persistence", "forgiveness" and "feedback" rather than "continuity", "reversibility" and "visibility").

However the main part of these style guides is concerned with using the functions of the user interface library to make applications look part of a family. The Macintosh guidelines include precise instructions on shape and size of buttons. The Windows guidelines describe the standard appearance of windows, menus, icons and toolbars as well as internal interfaces such as use of the registry or OLE. These standards do have some implications for usability - they make it easier for users to recognise the behaviour of certain types of controls - but they do not address many important aspects of usability. However they sometimes offer contradictory advice. Suns' Open Look guidelines recommend placing menu items in order of the process being carried out, but elsewhere say that most frequently used items should be at the top. Finally, style guides have an increasing function of providing corporate branding for the operating system suppliers, and this sometimes overshadows usability concerns.

1.4. Heuristic evaluation

If a programmer has more ambition than wanting to create a generic Windows (or Macintosh or Motif) application, how can he or she apply the lessons from the years of experimentation with different interfaces? One approach is to collect all the lessons that have been learned through different stages of hardware development, and then test in a systematic way whether our new design takes those lessons into account. This first theme has given a review of the most important of these historical discoveries - they include the use of command languages, dialogues, WYSIWYG, menus and direct manipulation. Nielsen suggested that the usability of a system should be evaluated by a panel of experts, each working from a list of usability *heuristics*. Every aspect of the system function could then be compared to the things we know about usability, and the results aggregated in order to direct the system design. The practical arrangements for conducting heuristic evaluation, together with a suggested set of heuristics, are covered in the first of the analysis technique descriptions later in these notes.

Lecture 3 starts about here -

Theme 2 - Psychological User Models

2.1. The need for user models

The first theme described the historical development of computer user interfaces, and the way that this has provided us with a set of design guidelines and heuristics for developing usable software. It hinted that this development involved psychologists who analysed the requirements of human users, but gave no details of the type of research conducted by those psychologists. A great deal of this research was actually conducted here in Cambridge, after the establishment of the Applied Psychology Unit in Chaucer Road. The APU was created in response to problems observed during the second world war - as military hardware became more technical, its performance was often limited by the abilities of the operators (e.g. gunners, pilots or tank drivers) rather than specification of the machines. The APU recruited researchers from experimental psychology who could apply earlier discoveries about human perception and movement, and who also conducted experiments on volunteers from a nearby military establishment.

This concern with machine interaction resulted in the borrowing of theory from the machine engineers, applying it to the performance of the machine operators. The engineering theories of closed loop control and information transfer described complex systems in terms of the dynamics of interacting subsystems. Research into usability therefore described the operators themselves as further *black box* subsystems, having inputs (observing the world, and the state of the machine), and outputs (controlling the machine). In order to understand the dynamic performance of the whole system, it was clearly necessary to understand the human portion of the system as well as the machine. Some aspects of these communication channels, such as *visual perception* of simple shapes and lights, had been studied for over 50 years. Others required new research which was conducted in Cambridge and at the other research centers creating the discipline of *cognitive psychology*. The set of black boxes that have resulted bear a rather close resemblance to generic computer architectures - they include models of visual input, of physical output, of memory, and of a problem-solving processor that uses that memory for intermediate results.
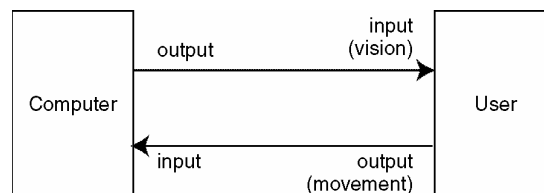
2.2. Models of visual input

The basic characteristics of human vision are highly important to computer graphics, because the ultimate goal of graphics is creating a visual stimulus sufficient for human observers to see a coherent and meaningful display. Some of this material was therefore covered in the Part 1B Computer Graphics and Image Processing course. The Part 1B course described the structure of the human *retina*, which is composed of brightness sensing *rods* and colour sensing *cones*. The cones are concentrated in the *foveal* region, which provides finer resolution of detail. The characteristics of colour vision can be described in terms of *colour spaces* that quantify and linearise the subjective impressions resulting from different levels of stimulation of red, blue and green sensing cones. Our eyes do not simply register colour planes, however. They *adapt* to

different levels of brightness, are sensitive to local **contrast**, and impose **quantisation** over physically uniform intensity distributions.

2.2.1. Marr's theory of vision

These results do not give us a great deal of information about how we interpret the signals coming from our eyes, and are thus of limited use for describing interaction with machines (beyond our ability to observe flashing coloured lights). David **Marr** proposed a series of black boxes which were involved in interpreting this visual input. The first level is the retinal image. This is processed (by nerve cells in the retina) to find the boundaries between relatively uniform regions. The result of this edge filtering is a **primal sketch**. The structure of the primal sketch can be analysed to form a **2½D sketc**h in which 2D regions are identified as being in front of or behind other regions. Finally this intermediate representation is resolved into a **3D model** of the object being perceived.

Marr's theory of vision can be regarded as the inverse of the computer graphics process in which 3D models are converted into two dimensional surfaces and edges for rendering. It provides us with a basis for simulating and modelling the process of understanding visual displays. It is certainly adequate for describing the way that we interpret window displays, which are essentially 2½ dimensional. The use in display windows of uniformly coloured regions together with shadows to disambiguate stacking order means that we can interpret these as being meaningful objects.
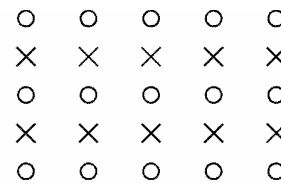
2.2.2. Gestalt laws of perception

Marr was more concerned with perception of the physical world than with two dimensional displays, but many interesting characteristics of display interpretation had already been established by German Gestalt psychologists before 1920. These **gestalt principles** of visual organisation include the principles of:
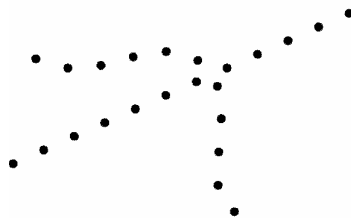
- **Proximity** - elements close together tend to organise into units
- **Similarity** - objects that look alike tend to be grouped together
- **Good continuation** - we see lines as being continuous if they do not bend sharply
- **Closure** - we prefer to see regular shapes, inferring occlusion to do so
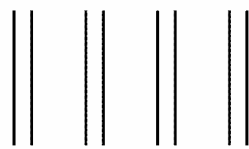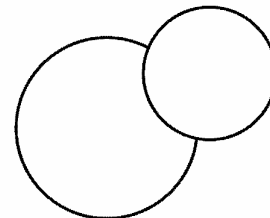
Proximity

Similarity

Good continuation

Closure

Gestalt laws of perception are clearly critical to graphic design, and are important characteristics of a two-dimensional user interface.

2.2.3. Visual tasks

There has been substantial research into human performance for basic visual tasks. Many of these results are relevant to perception of displays, but are too detailed for an introductory course. Some relevant topics that are covered at length in specialist textbooks include:

Depth perception - our ability to perceive distance in the physical world relies on many different perceptual factors. The basic mechanism of depth perception, binocular stereo vision, is not very reliable, so we supplement it with a wide variety of monocular depth cues. On current computer displays these cues are the only means of depth perception, as the display is rendered on a flat screen.

Face recognition - we are able to carry out certain types of image segmentation very quickly. Processing of human faces within a scene occurs far more quickly than other types of scene analysis, and we can perform some high-precision tasks such as identifying direction of gaze with great accuracy.

Visual search - when we need to identify one object among a field of different objects, several effects are relevant. Finding a given letter in a list of letters varies linearly with the length of the list, but we can find a differently coloured letter in near-constant time. Other visual "pop-out" effects include the speed with which we can find local variations in brightness, or identify a shape with a different orientation in a field of identical shapes. One particular measure of visual search was discovered at the Cambridge Applied Psychology Unit by a researcher working on gun aiming: Hick found that the time to find one item among a number of similar items (i.e. discounting pop-out effects) is related to number of items by a log function, now called ***Hick's Law***:

$$T = k \log_2 (n + 1)$$

2.3. Models of physical "output"

Psychological studies of mental "outputs" have typically concentrated on speech more than physical action. However human speech is currently less relevant to HCI, as long as speech-operated interfaces are relatively uncommon. The dynamics of reaching actions are also relatively well-understood, however, because they are mechanically quite predictable. We know that reaching out to pick something up involves a high speed ***approach*** phase followed by a slower ***homing*** phase. While moving, our hands already form the appropriate shape for ***grasping*** the object.

This kind of study is less relevant to current user interfaces, however. Most interesting are studies of typing, and studies of pointing. Studies of typing allow us to investigate some urban myths of the user interface. For example, the "qwerty" keyboard is not so inefficient as many computer scientists believe. People can make successive strikes more quickly with alternate hands than with the fingers of the same hand. The qwerty keyboard happens to facilitate this quite well, so that even non-typists (in 1981 - before keyboards were as widespread as today) could type far more quickly on a qwerty keyboard (1.08 letters/s) than an alphabetic one (0.65 letters/s).

The process of pointing at a target was of particular interest when mice were first introduced. How was it possible to compare the speed of pointing at something with a mouse to the speed of typing a command, or making a series of cursor movements? Fortunately, this is another result that was established during early research on machine models of human motion. ***Fitts' law***

describes the fact that the time it takes to point at a given location is related to the size of the target and to the distance from the current hand position to the target.

Fitts experiment involved two targets of variable size, and separated by a variable distance. Experimental subjects were required to touch first one target, then the other, as quickly as they could. The time that it takes to do this increased with the *Amplitude* of the movement (i.e. the distance between the targets) and decreased with the *Width* of the target that they were pointing to:

$$T = K \log_2(A / W + 1) \qquad \text{where A = amplitude, W = width}$$

Fitts' law has been found to apply quite well to pointing operations with a mouse, and can hence be used to predict human performance when using menu interfaces.

2.4. Models of memory

One of the most famous findings in cognitive psychology research, and the one most often known to user interface developers, is an observation by George Miller in 1956. Miller generalised from a number of studies finding that people can recall somewhere between 5 and 9 things at one time - usually referred to as "*seven plus or minus two*". Surprisingly, this number always seems to be about the same, regardless of what the "things" are. It applies to individual digits and letters, meaning that it would be very difficult to remember 25 letters. However if the letters are arranged into five 5-letter words (apple, grape …), we have no trouble remembering them. We can even remember 5 simple sentences reasonably easily. Miller called these units of short-term memory *chunks*. It is rather more difficult to define a chunk than to make the observation - but it clearly has something to do with how we can interpret the information. This is often relevant in user interfaces - a user may be able to remember a sequence of seven meaningful operations, but will be unable to remember them if they seem to be arbitrary combinations of smaller elements.

Short term memory is also very different from *long term memory* - everything we know. Learning is the process of encoding information from short term memory into long term memory, where it appears to be stored by association with the other things we already know. Current models of long-term memory are largely based on *connectionist* theories - we recall things as a result of activation from related nodes in a network. According to this model, we can improve learning and retrieval by providing *rich* associations - many related connections. This is exploited in user interfaces that mimic either real world situations or other familiar applications.

A further subtlety of human memory is that the information stored is not always verbal. Short term memory experiments involving recall of lists failed to investigate the way that we remember visual scenes. Visual working memory is in fact independent of verbal short term memory, and this can be exploited in mnemonic techniques which associate images with items to be remembered.

2.5. Models of problem solving

Basic models of human problem solving are completely familiar to computer scientists who have studied research into artificial intelligence. The earliest models of problem solving in cognitive psychology are derived from the 1969 work of Ernst and Newell on a *Generalised Problem Solver*. The GPS operated in a search space characterised by possible intermediate states between some initial state and a goal state. Problem solving consisted of finding a series

of operations that would eventually reach the goal state. This involved recursive application of two heuristics: A) select an intermediate goal that will *reduce the difference* between the current state and the desired state, and B) if there is no operation to achieve that goal directly, *decompose it into sub-goals*.

This model of problem solving as a recursive *hierarchy of sub-goals* has been widely adopted as a basis for the analysis of human problem solving. In physical tasks, the basic operations (and hence the leaves of the sub-goal tree) are physical actions. The required difference reduction can be deduced from perceptual (visual) input. The difficulty of the problem can be analysed in terms of the depth of the sub-goal tree; if the tree is too deep, the limited capacity of working memory will result in a "stack overflow" so that we forget what we should do next to finish the problem.

2.6. System models for HCI

2.6.1. The Model Human Processor

These results from psychological research give us the basic elements that we need to describe the whole interaction loop of the computer and its user. We simply need to combine the observations that have been made regarding performance of the black boxes for visual perception, motion, and cognitive processing. This was precisely the approach taken by Card, Moran & Newell, who proposed a minimal *Model Human Processor* for predicting the speed with which users could carry out tasks on a computer. They assumed that the task could be broken down into unitary *perceptual* events, *motion* events, and *cognitive* events, and that the total time required to carry out a sequence of events could be estimated by adding the time usually required for each type of event. Based on earlier studies, they observed that perceiving a stimulus takes somewhere between 50 and 200 ms (typically 100), that making a simple decision takes somewhere between 25 and 170 ms (typically 70), and that making a tapping motion takes somewhere between 30 and 100 ms (typically 70). They proposed a number of other elaborations, but the essential parameters are:

$$t_p = 100 \ [50 \sim 200] \ \text{ms}$$

$$t_c = 70 \ [25 \sim 170] \ \text{ms}$$

$$t_m = 70 \ [30 \sim 100] \ \text{ms}$$

The total time required for some user interface action is then predicted by the number of times each type of event must occur in the performance of that action.

$$T = n_p t_p + n_c t_c + n_m t_m$$

2.6.2. Keystroke Level Model

Card, Moran & Newell refined their model for use as a system design tool that could be used by designers to compare the operation speed of alternative design. The keystroke-level model added operators to describe specific types of movement - *keystrokes*, *pointing* with a mouse and *drawing*. They added the system *response* time and the time to make mental *preparations*, and defined the circumstances in which mental preparation would be required. The keystroke level model is described in more detail in the second of the analysis technique sections later in these notes.

2.6.3. GOMS

The keystroke level model predicts only basic actions, not complex task procedures. The GOMS model (Goal, Operators, Methods and Selection) extended KLM further, with a detailed model of problem solving based on the Newell's earlier work on the General Problem Solver. The *operators*, or basic actions taken by the user, corresponded to the components of the keystroke level model. The rest of GOMS described the process of selecting operators in order to accomplish a complex task. An experienced user would be expected to have some repertoire of *methods* - sequences of operations that are known to accomplish a particular goal. Where more than one method can be used to achieve the same goal, the model predicted that extra time would be required for *selection* - choosing the best method to use. GOMS attempted to provide quantitative models of the process of goal hierarchy decomposition, the working memory required to store goals and working values, and also the learning processes that are involved in acquiring new methods.

The implication of the GOMS method was that the requirements of the user can be predicted exactly from the nature of the software application. It should be possible to predict exactly what actions the user will take in advance by simulating their reasoning with the GOMS model. Having done that, the user interface could simply be designed to provide the most efficient path through the goal hierarchy. As it happens, this theoretical goal seems to have been too ambitious. GOMS provided a very neat fit of a psychological model to software design, but this convenience can be attributed partly to the fact that the GPS model of human problem solving was itself a computer program in the first place. At the level of complexity of real user interfaces, the behaviour of early AI simulations like GPS bears little resemblance to the behaviour of real humans.

2.6.4. Recent complex models

HCI research is continuing to develop more complex models of human cognitive processes, in the hope that they can be used to predict usability through automated analysis. This course will not give any further consideration to *programmable user models*, which are generally based on the latest generation of computer models of human cognition. The Applied Psychology Unit in Cambridge was a major centre of this type of work in the 1980s and early 1990s. Richard Young's work on Programmable User Models continues at the University of Hertfordshire.

Phil Barnard, still at the Applied Psychology Unit (now Cognition and Brain Science Unit), is developing a cognitive model that describes cognitive processes beyond the domain of HCI. His *Interacting Cognitive Subsystems* model describes human cognitive processing in terms of nine cognitive subsystems having a common structure. Jon May has used the ICS model to describe interaction processes including attention within a display, different levels of mental representation and the interaction of knowledge with task structure. He proposes that usability can be analysed by comparing changes of state in task structure to mental structure using transition path diagrams. This method of cognitive task analysis is described in a paper at:

    http://www.shef.ac.uk/~pcljm/papers/ICS.NATO.pdf

Theme 3 - Evaluation of Cognitive Models

The second theme described the many attempts to construct exact analytical models of the human user, in order to make precise mathematical estimates of the time required to operate a user interface. These models are based on empirical data from psychological experiments, but the granularity of behaviour that is described has been at such a low level that it can only be used to analyse very simple and uninteresting actions. As a result, HCI practitioners have adopted a set of observational, experimental and design techniques that are closer to the level of granularity we normally use when discussing complex human tasks.

3.1. Mental models

Mental models research attempts to describe the structure of the mental representations that people use for everyday reasoning and problem solving. Common mental models of everyday situations are often quite different from scientific descriptions of the same phenomena. They may be adequate for basic problem solving, but break down in unusual situations. For example, many people imagine electricity as being like a fluid flowing through the circuit. When electrical wiring was first installed in houses, it appeared very similar to gas or water reticulation, including valves to turn the flow on and off, and hoses to direct the flow into an appliance. Many people extended this analogy and believed that the electricity would leak out of the light sockets if they were left without a lightbulb. This mental model did not cause any serious problems - people simply made sure that there were lightbulbs in the sockets, and they had no trouble operating electrical devices on the basis of their model.

The psychological nature of unofficial but useful mental models was described in the 1970s, and these ideas have been widely applied to computer systems. Young's study of calculator users in 1981 found that users generally had some cover story which explained to their satisfaction what happened inside the device. Payne carried out a more recent study of ATM users, demonstrating that even though they have never been given explicit instruction about the operation of the ATM network, they do have a definite mental model of data flow through the network, as well as clear beliefs about information such as the location of their account details.

The basic claim of mental models theory is that if you know the users' beliefs about the system they are using, you can predict their behaviour. The users' mental models allow them to make inferences about the results of their actions by a process of ***mental simulation***. The user imagines the effect of his or her actions before committing to a physical action on the device. This mental simulation process is used to predict the effect of an action in accordance with a mental model, and it supports planning of future actions through inference on the mental model. Where the model is incomplete, and the user encounters a situation that cannot be explained by the mental model, this inference will usually rely on ***analogy*** to other devices that the user already knows.

If we were to describe this in terms of computational problem solving models, we have to recognise that users are working simultaneously in two different problem state spaces. One of the spaces describes the actual structure of the users' goals, which may not explicitly recognise the computer at all. The other describes their understanding of the device state space. These two state spaces are ***yoked*** together, so that moves in one state space can only be accomplished by equivalent moves in the other. If we use this model to analyse usability, we must also recognise that the user's model of the device will not be the same as the designer's model of the device - mental model theory attempts to construct an explicit characterisation of the user's model as a basis for design.

It should be clear that mental models theory is far more complex than the Model Human Processor and GOMS theories described in the second theme. Although the user model is very likely different to the designer's model, and may be "unscientific", it can still be quite complex, with a complex set of mappings between goals and device that may be different for different users. It is very difficult to make a predictive computational simulation of the mental model and its effects on users' problem solving. Instead, we must study the behaviour of users. We assume that system users generally have good reasons for their actions, and orient our research toward finding out what those reasons are.

3.2. User-oriented design methods

The remainder of this theme describes some of the approaches that have been applied to the study of system users for usability purposes.

- Prototyping

- Experimental studies – finding patterns within variation

- Hypothesis testing, questionnaires, think-aloud protocols

- Cognitive walkthrough

3.2.1. Prototyping

Prototyping is becoming increasingly important as a software design method, particularly addressing the problems of developing user interfaces within a strict *waterfall* development model. Companies that use waterfall models have placed increasing emphasis on accurate portrayal of the user interface at the specification phase, after finding that the majority of specification changes arise from client not understanding the requirements for user functionality. In terms of mental models theory, this could be expected - clients who have no image of the interface that they will operate are unlikely to have a useful mental model of system behaviour.

If the system can respond in complex ways, it is difficult to appreciate this from static figures in a specification, so the specification phase of projects often uses *rapid prototyping* tools to construct a functional user interface. This prototype can be demonstrated to clients and used as a basis for discussion. If a spiral development model is adopted rather than a waterfall, the prototype can be *refined iteratively* until the full system functionality is achieved. Incremental prototyping requires that the rapid prototyping tool also meets the engineering requirements of the final system. If such a tool is not available, an alternative is *deep prototyping*, in which one aspect of the system functionality is fully implemented before developing the rest of the interface.

These common approaches to prototyping are quite different to the prototyping techniques that have been found to be successful in developing novel user interfaces. Many product designers believe that creativity in the product design process is directly related to the number of prototypes produced. HCI research similarly emphasises techniques for developing a large number of prototypes, exploring different possible solutions, and evaluating the usability of alternatives. This is in contrast to incremental prototyping techniques, which encourage cost-saving by using the first solution regardless of its usability properties.

Investigation of multiple prototypes requires low cost techniques for producing prototypes. Rather than implementing realistic system functionality, these often use generic graphic design tools with some scripting functions: early HCI research often used Apple Hypercard, and more recent work uses Macromedia Director. An even more radical proposal is *low-fidelity*

prototyping, in which the prototype user interface is made using controls built from glue and paper. During evaluation, the functionality can be implemented using the ***Wizard of Oz*** technique - a person simulates the machine by responding to user actions with the display of new (paper) screens.

The objective of building multiple prototypes is to investigate design alternatives through evaluation with actual users. The next section describes some techniques for making this evaluation.

3.3. Evaluation techniques

3.3.1. Controlled experiments

The most common empirical method used in HCI is the controlled experiment. An experiment is based on a number of ***observations*** (measurements made while someone is using an experimental interface). A typical measurement might be "How long did Fred take to finish task A?" or "How many errors did he make"? A wide range of alternative measurements are possible, including heart rate or other exotic biological data. However we most often assume, as in the discussion of KLM and GOMS, that it is a good thing if interfaces allow us to do something quickly.

A single observation of speed is not very interesting, however. If Fred did the task again, he would take a different amount of time, and if someone else did it, it would take an even more different amount of time. We therefore collect sets of measurements, and compare averages. The sets might be multiple observations of one person performing a task over many ***trials***, or of a range of people (experimental ***subjects***) performing the same task under controlled conditions. As with most human performance, the measured results will usually be found to have a ***normal distribution***.

A typical HCI experiment involves one or more experimental ***treatments*** that modify the user interface. A very simple example might test the question: "How long does Fred take to finish task A when using a good UI, compared to a bad UI?" The result will often be that the good UI is *usually* faster to use than the bad, but not in *every* trial. If we plot the measurements, we find two overlapping normal distributions, and we must therefore compare the effect of treatments relative to the spread in the population distribution. We need to know whether the difference between the averages is the result of normal random variation, or the effect of the changes we made to the user interface.

This involves a statistical ***significance test*** such as the ***t-test***. The t-test and other similar tests answer the question "*What is the probability that the observed difference in means is due to random variation*?". This is called the ***null hypothesis***, and we generally hope that the answer will be "*the probability is very low*" - i.e. that the observed difference is most likely because we designed a really good interface. In HCI research, we usually insist that the probability of the result being due to random variation (p) is less than 0.05, or 5%. Good quality research results are normally based on experiments with significance values $p < 0.01$.

Sources of variation in reaction time

Some people find it surprising that we can draw scientific conclusions from measurements that are different every time we make them. This is rather fatalistic. We all agree that people are different. If there were no way to measure the value of a user interface for a wide range of

different people, there would be no chance of progress in user interface development. It is important, however, that we are aware of the sources of variation in the measurements.

These include:

- Variations in the task performed;
- The effect of the treatment (i.e. the user interface improvements that we made);
- Individual differences between experimental subjects (e.g. IQ);
- Different stimuli for each task;
- Distractions during the trial (sneezing, dropping things);
- Motivation of the subject;
- Accidental hints or intervention by the experimenter;
- Or other random factors.

The statistical techniques used in sophisticated experiments isolate these kinds of factors, and try to account for them separately in order to gain a good understanding of the effects of the experimental treatments. Fortunately over a large number of trials all of these factors tend to combine into a pattern of random variation within the normal distribution, as predicted by the ***central limit theorem***. The central limit theorem and further null-hypothesis testing techniques are beyond the scope of this course. A useful introductory text on experiment design is Robson's *Experiment, Design and Statistics in Psychology*. A briefer summary of the most important principles is given in section 14.4 of Preece, Rogers and Sharp.

A more serious concern in this kind of research is the ***validity*** of the result. Would the observed effect generalise to other situations besides the precise context of the experiment? What exactly was the mechanism by which the effect occurred? Is there some established HCI work or psychological theory that can explain it? Could it be ***replicated*** if you repeated the experiment with slight variations (older users, for example, or a different model of computer)? In order to avoid these potential criticisms, HCI researchers often try to use experimental tasks and context that have good ***environmental validity*** - they are as close as possible to the situation in which the interface will really be used.

3.3.2. Other empirical techniques

Hypothesis testing is a very useful technique for making quantifiable statements about improvements in a user interface. It also hides a lot of useful information, however. Experimental subjects usually have a lot of useful feedback about the interface that they are trying, but there is no easy way to incorporate this into statistical analyses. Instead, we use a range of other techniques to capture and aggregate interpretative reports from system users.

Surveys

Surveys include a range of techniques for collecting report data from a population. The most familiar types of survey are public opinion polls and market research surveys, but there are a much greater range of survey applications. Surveys are usually composed of a combination of ***closed*** and ***open*** questions. Closed questions require a yes/no answer, or a choice on a ***Likert*** scale - this is the familiar 1 to 5 scale asking respondents to rank the degree to which they agree with a statement. Closed questions are useful for statistical comparisons of different groups of respondents. In open questions the respondent is asked to compose a free response to a the question. The latter requires a methodical ***coding*** technique to structure the content of the responses across the population, and is particularly useful for discovering information that the investigator was not expecting.

Questionnaires

Questionnaires are a particular type of survey. (Interview studies of a sample population are also a form of survey). Questionnaires are generally used to gather responses from a larger sample, and can be administered by email as well as on paper. A discussion of the issues that can be encountered in questionnaire studies is available on-line at: http://kmi.open.ac.uk/people/paulm/summer98/question.html.

Think aloud studies

Much cognitive psychology research, including some basic research on mental models, is based on **think-aloud** studies, in which subjects are asked to carry out some task while talking as continuously as possible. The data are collected in the form of a **verbal protocol**, normally transcribed from a tape recording so that subtle points are not missed. Use of this technique requires some care. It can be difficult to get subjects to think aloud, and some methods of doing so can bias the experimental data. A detailed discussion of this kind of study is provided by Ericsson & Simon (1985).

Bad techniques

Some user interface developers use evaluation techniques that are practically useless. Unfortunately these techniques can even be found in some published research in computer science. This section is included as a warning to interpret such results with great care.

Simple **subjective reports** seldom give useful information about interface usability. When users are shown a shiny new interface next to a tatty old one, they will often say that they like the new one better, regardless of its usability. There are many circumstances in which a person's introspective feelings about their mental performance is not a good predictor of actual performance, so this type of report is unreliable as well as open to bias.

Some research proposes a usability hypothesis, then does not test it at all. "It was proposed that more colours should be used in order to increase usability". This type of statement is speculation rather than science; designing novel user interfaces without any kind of experimental testing is rather pointless.

There is a great deal of variation between different people in their ability to use different interfaces. This may result from different mental models, different cognitive skills, and many other factors. Any conclusions drawn from an observation of only one person must therefore be very suspect. Unfortunately, many user interfaces are developed based on observations of a single person - the programmer. The **introspection** of the user interface developer about his or her performance is seldom relevant to users.

The word "intuitive" is often used in discussion of user interfaces to summarise theories based on all the above.

3.4. Cognitive walkthrough

Cognitive walkthrough is an evaluation technique that incorporates a more sophisticated theory of user behaviour, incorporating an implicit user model and a theory of exploratory learning (Lewis and Polson's CE+ theory) based on empirical studies. Cognitive walkthrough is described in more detail in the third of the analysis technique sections later in these notes.

Theme 4 - Task-Oriented Analysis

This theme returns to the topic of the first, in that it is oriented more toward engineering than toward psychology. In fact it borrows techniques from other social sciences that have been found to be useful in software design.

4.1. Observation and task analysis

4.1.1. Structured interviews

Most software projects start with a series of meetings in which the system requirements are established. The agenda of these meetings is often concerned with many other matters than the user interface, however. In fact the people who will use the completed system may not even be present. Their requirements are defined by a representative (a system analyst for an internal project or a market researcher for a product) who may not have much experience of design for usability.

For this reason, user interface designers often conduct studies specifically to discover the requirements of the system users. One of the cheapest and most straightforward techniques is to conduct *interviews* with the users. Interviews must be carefully planned to be effective, however. They are generally more or less *structured*, encompassing a selected range of users, and taking care to encourage cooperation from users who may feel threatened or anxious.

A structured interview is based around a set of questions that will be asked of every interviewee. This need not necessarily be a long list, but it helps to collect data into a common framework, and to ensure that important aspects of the system are not neglected.

Chapter 13 of Preece, Rogers and Sharp gives far more detail about interview techniques.

4.1.2. Observational studies

Observational studies are a less intrusive way of capturing data about users' tasks, and can also be more objective. They involve more intensive work, however. An observational study of tasks that take place in a fixed location can be conducted by making video recordings which are transcribed into a *video protocol*. This protocol can then be used for detailed analysis of the task - relative amounts of time spent in different sub-tasks, common transitions between different sub-tasks, interruptions of tasks and so on.

Audio recordings can also be used for this purpose in certain domains, but these are less likely to be useful for task analysis than they are in think-aloud experiments.

If a task ranges over a number of locations, the investigator has no choice but to follow the subject, taking notes or recordings as best as possible. This is sufficiently difficult that *ethnographic* techniques are more likely than passive observation An alternative is the user of diary studies, in which subjects take their own notes, but prompted to pay attention to specific times, events or categories.

4.1.3. Ethnographic field studies

Ethnographic study methods recognise that the investigator will have to interact directly with the subject, but while taking sufficient care to gain reasonably complete and objective information. An ethnographic study will attempt to observe subjects in a range of contexts, over a substantial period of time, and making a full record using any possible means (photography,

video and sound recording as well as note-taking) of both activities and the artefacts that the subject interacts with.

Ethnographic methods are becoming increasingly important in HCI, to an extent that many technology companies will now employ an anthropologist as their first social science expert, rather than a psychologist. In practice, both sets of skills are useful. Cognitive descriptions of human performance (often called **human factors** by engineers) tend to be most valuable in detailed assessment and critique of a proposed design. Descriptions of mental models can be helpful in elaborating a design concept. But ethnographic observation can help to understand technology and products in completely new ways, perhaps leading to innovative new concepts. In this respect, ethnography can be considered as a contribution to engineering **requirements capture** in a traditional technology company. Younger and trendier companies like to describe the whole process of product concept identification, development and refinement as **user experience** (UX) design. There are specialised books and conferences that report methods and research from all of these perspectives (e.g. EPIC: the Ethnographic Praxis in Industry Conference; DUX: Designing the User Experience; CHI: Human Factors in Computing, etc.).

HCI researchers tend to have skills in all these techniques, but product designers generally want a simpler recipe that doesn't require them to spend a year or more doing fieldwork. Often the biggest problem they have is how to gain a perspective of what it is like to be a user, escaping the mindset of their own technical understanding and expectations of the product. A useful intermediate technique is to write fictionalised descriptions of the kind of person who will use the product, to help the engineer understand what sort of person they are based on his or her personal experience. These user **personas** might be derived from ethnographic fieldwork, or from conventional market research data. They are a particularly popular technique in Microsoft, where persona descriptions include photographs (presumably of actors), fictional biographies, and descriptions of why this person uses computers. Product design then proceeds on the basis that the designer tries to accommodate (or ideally charm, assist and delight) this range of fictional people. Two Microsoft staff members have written a brief paper explaining their use of the technique (Pruitt and Grudin – see bibliography).

Chapter 12 of Preece, Rogers and Sharp gives far more detail about ethnographic and observational techniques.

4.1.4. Field tests

Some very successful products have carried out **field testing** of their products in addition to field studies at the specification phase. A well-documented example is the "follow-me-home" programme carried out by Intuit Inc. after the release of their Quicken product. Company researchers selected customers at random, when they were buying a shrink-wrapped copy of Quicken in a store. The researcher then went home with the customer in order to observe them as they read the manuals, installed the product, and used it for their home financial management. Intuit directly attribute the impressive success of the product to this type of exercise, and to the observational studies they carried out during initial product planning. (Quicken survived an assault from a Microsoft product priced at a predatory $15, and Microsoft later made a bid of $1.5 billion to buy Intuit).

4.2. Use case design

User-centred analysis and design was once considered a fairly radical approach to software development, and books on the topic were more likely to be written by HCI researchers than by software engineers. **Software design** is now recognised as an important discipline with objectives that differ from the main objectives of computer science research. Software design is

ultimately concerned with the needs of users, for reasons that were argued in the introduction to these notes.

This emphasis on system users when designing software is now recognised in the expanding use of object-oriented design methods based on the Universal Modelling Language *UML*. UML was created through the synthesis of several earlier design methods, including Ivar Jacobson's Object Oriented Software Engineering. The OOSE method prescribed the analysis of user activities in terms of *use cases* - specific scenarios for interactions with the system.

The use cases from OOSE have been adopted completely into UML. Use case analysis is the first stage of system design with UML, in which the behaviour of the system is described from the point of view of abstract *actors*. Actors represent abstract roles that users will take when interacting with the system, potentially structured according to classes of users. A use case is a narrative of some specific interaction that a specific actor conducts with the system.

Use cases are relevant through later stages of system design, as they can be used for the specification and validation of event traces involving different objects and subsystems. They provide sufficient formality for this description of system behaviour, but also provide a comprehensible unit of user interface functionality that can be discussed directly with clients and users. These attributes theoretically provide *traceability* of system implementation from specification through to the object-oriented design, and should therefore allow straightforward modification of the system when use cases are altered in response to maintenance requirements.

Various international research efforts are in place to integrate use case oriented design with more psychological approaches to HCI. At the time of writing, substantial progress is still awaited.

4.3. Cognitive dimensions of notations

The approaches to user-oriented design that have been described in this theme are rather atheoretical when compared to the psychological theories and methods in the previous themes. This is one of the main challenges for HCI - to integrate theoretically sound cognitive models into engineering design processes.

That is the aim of Green's *Cognitive Dimensions of Notations*, developed from 1989 onwards at the Applied Psychology Unit in Cambridge, and now at the Computer Based Learning Unit in Leeds. The cognitive dimensions are designed for use in an environment where designers accept that there can be no perfect user interface. As in all fields of engineering, every user interface design is a compromise. Even if a user interface were constructed that was perfectly suited to a particular user carrying out a particular task, it would not be perfect for other users and tasks. The cognitive dimensions therefore aim to provide designers with a working *vocabulary* in which they can discuss usability issues that are cognitively relevant while also being recognisably related to the potential solutions. The dimensions are partially independent, in a way that means *trade-offs* can be analysed, discussed and selected as appropriate for a particular design.

The cognitive dimensions framework describes the system under investigation as an *information artefact* - something that has been built for the processing, storage and communication of information. Every information artefact provides one or more *notations* in which the information being manipulated is encoded. The notation itself does not uniquely determine usability, however. The environment used to manipulate the notation is equally

important. The complete system of the notation and the environment can be analysed to determine its usability for different tasks. This analysis process, and the dimensions themselves, are described in more detail in the last of the analysis technique sections later in these notes. There is a chapter in Carroll that presents this material in an expanded, textbook format, with more background research.

4.4. Research trends in HCI

Research into HCI is an active field. Reports of recent research can be found in the annual proceedings of the ACM CHI conferences called *Human Factors in Computing Systems*, and in a range of specialist journals including *SIGCHI* publications, the *International Journal of Human-Computer Studies*, *Human-Computer Interaction*, *Behaviour and Information Technology* and others.

There are several important sub-fields which have expanded sufficiently to have their own conferences and research groups. Some examples of these significant areas are:
- Computer supported cooperative work
- Information appliances / Ubiquitous computing
- New interaction devices
- End user programming
- And more …

4.4.1. Computer supported cooperative work

Traditional HCI research has focused on a single user sitting in front of a computer, and has neglected the environment that the user is working in. Most users of complex computer systems do not work alone, but in organisations where they must cooperate with many other users. **CSCW** research investigates how the user interface can support this collaboration.

One stream of CSCW relies on a structured analysis of human collaboration, using software to organise online discussion into this structure. Design discussions, for example, can often be broken down into a series of **questions** that need to be addressed, **options** for addressing each question, and **criteria** by which an option should be selected. The discussion can be structured by graphical presentations showing the relationships between the questions, options and criteria. This type of argumentation support system can potentially be integrated into computer-aided software engineering tools to provide a record of **design rationale** - the reasons why design decisions were taken.

It is possible to conduct basic research into CSCW using only networked workstations, but there are many more sophisticated alternatives. A great deal of research has been conducted into uses of video conference technology and networked **shared whiteboards**, which provide a common workspace for physically separated groups. This research analyses the interaction between groups of people working together in order to determine which attributes need to be preserved in physically distributed collaboration.

An alternative approach to CSCW research analyses the interaction in successful **online communities**. Examples include **Usenet newsgroups** (ucam.cl.students appears to be an effective networked community), and collaborative role-playing environments such as **multi-user dungeons** (MUDs). Graphical **avatars** and **shared virtual reality** promise to provide interesting developments in this area.

4.4.2. Information appliances / Ubiquitous computing

As microprocessors are incorporated into a greater range of devices, it is reasonable to ask what potential this offers for new modes of interaction with both new gadgets and traditional appliances. Research groups working in this field (including several in Cambridge) investigate what you can do by incorporating computing and communication functions into ordinary appliances such as toasters, kettles, refrigerators, desks, televisions or radios.

Large commercial laboratories also investigate the potential of expanding the range of computing devices, and integrating a wider range of information devices into our environment. There are many opportunities to expand the capabilities of existing devices such as PDAs, cell-phones or smart cards, as well as introducing new categories of device - intelligent walls and paper, smart badges, keyrings or jewellery. Much of this research currently concentrates on providing different styles of network interaction using a variety of devices that establish the user's location and identity, but have only minimal user interfaces.

Sometimes the physical form of the objects enables very different types of manipulation, in which physical objects are used to interact with the computer (perhaps wirelessly or by using video information to capture their position). This is described as a *tangible user interface*, and can be contrasted with conventional use of keyboards and mice, buttons, or stylus-based screen interaction.

4.4.3. New interaction devices

Hardware devices such as the mouse and the bit-mapped display have been extremely influential on the current generation of user interface, as was described in the first theme. There are continual attempts to define and characterise the next generation of interaction devices. Examples include extended capabilities of current hardware - new techniques for pen input (e.g. text entry techniques, gesture languages) or extended physical interaction (two-handed interaction techniques such as *information lenses*, foot-operated devices or *gaze tracking*).

Interaction in 3D spaces is still a very difficult problem. Standard hardware includes 3D position sensors such as head trackers, data gloves, and 3D mice or wands. The effectiveness of these devices differs depending on the context, the task, and the type of display - 3D display on a screen offers different opportunities from *immersive* virtual reality, or *augmented reality* in which data displays are superimposed on the user's view of the real world.

It is unclear how sound should be exploited in a user interface. As speech recognition technology improves, it is likely to require a completely new interaction paradigm - perhaps abandoning direct manipulation and returning to the command dialogue models of teletype-based interaction. There have also been occasional attempts to provide status information and feedback to system users by generating *non-speech audio* in response to user actions or system events.

4.4.4. End user programming

The study of programmers has always been an important sub-field of HCI, usually described as the *psychology of programming*. It was once the case that most computer users were likely to write at least small programs. The majority of computer users now do little programming, and this area of HCI has become less widespread, with specialist groups such as the Psychology of Programming Interest Group (PPIG) providing research reports. This topic is now gaining importance for user interfaces which include tasks that resemble programming: defining the behaviour of *agents* and *scripting languages*. This is often described as *end-user programming*, and there are many examples of commercial products providing programming

capabilities for specialist groups of users - the laboratory automation language LabView is one example. Macro languages in common desktop applications have clearly failed to provide benefits to most users. An alternative approach is ***programming by demonstration***, in which the system watches the user's actions, and infers a program which will automate the operations it observed.

Some large recent projects address the question of end-user software engineering, which are concerned with the problem that even non-professional programmers need to get their specifications right and debug the programs. This is a major problem in spreadsheets, for example, where research surveys routinely find large numbers of bugs, some of them with dramatic safety or business-critical consequences. Novel tools and techniques for testing, debugging, and formal specification are driving a great deal of original research into new spreadsheet technologies.

There has also been a great deal of research into special languages for ***teaching*** programming principles or general reasoning abilities. This is relevant to computer science students, but also to younger students including school-children and preschoolers. Psychologists and educationalists consider this type of research to be a high priority, and it is generally based on studies of actual students using existing and experimental programming languages in the classroom. Some of this research has resulted in commercial educational products such as Logo, ToonTalk, AgentSheets, StageCast Creator and Alice (details of all of these are available online).

4.4.5. Summary of Research Trends

Much HCI research is either developing or responding to new technologies, as has always been the case. Despite the fact that HCI research is often led by technology, the case made in the first theme of this course is still relevant. In research, as in commercial system development, usability can only be improved by the application of psychological and engineering principles. The techniques for usability analysis that have been described in these themes provide a basis for anticipating the usability implications of new technology, rather than simply implementing "neat" ideas with little knowledge about the effects they will have.

Analysis Techniques

Analysis Technique 1: Heuristic Evaluation

Nielsen suggested that the usability of a system should be evaluated by a panel of experts, each working from a list of usability *heuristics*. These are similar to Shneiderman's abstraction of the important principles of *direct manipulation* interfaces, including: Objects of interest should be continuously visible, operations should involve physical actions, effects should be rapid, visible, and reversible etc. Nielsen extended the list to other aspects of direct manipulation, and also to other features of the user interface besides direct manipulation.

However the heuristics themselves are not the most important feature of heuristic evaluation; they can be changed at any time. The essence of the technique is that once we have a set of such heuristics, it can be applied to make a *systematic evaluation* of a software system. The interpretation of the heuristics and the resulting evaluation are subjective, but the use of a panel of experts is intended to provide a degree of objectivity.

Procedure

It is therefore essential that heuristic evaluation involve *multiple evaluators* (preferably with differing backgrounds, in order to consider the system from different perspectives). Each evaluator inspects the interface alone, perhaps using a scenario describing the things that a typical user of the system would want to do with it. At each step of the inspection the evaluator compares its compliance to each of the heuristics - all evaluators use the same set of heuristics. The evaluators go through the interface at least twice, listing all the usability problems that they find. The results from all the evaluators are then compiled into a list documenting the usability problems of the system.

Sample heuristics

Nielsen provides a sample list of heuristics that might be used in an heuristic evaluation. A number of these are recognisably derived from principles of direct manipulation, although they apply to a wide range of different interaction styles.

Visibility of system status
> The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Match between system and the real world
> The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

User control and freedom
> Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards
> Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention
> Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

Recognition rather than recall
> Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use
> Accelerators -- unseen by the novice user -- may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Aesthetic and minimalist design
> Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Help users recognize, diagnose, and recover from errors
> Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Help and documentation
> Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Applicability

At the time of writing, heuristic evaluation is the most popular technique for assessing usability of software designs. It is simple and cheap to conduct, and appears easily justifiable on commonsense grounds. The disadvantages are that it provides little opportunity to address deeper system design problems, and that it does not provide any systematic way to generate solutions to the problems that are discovered.

More information on heuristic evaluation is available from Nielsen's web site:

```
http://www.useit.com/papers/heuristic/heuristic_evaluation.html
```

Analysis Technique 2: Keystroke Level Models

The keystroke level model (KLM) is a technique that is superior to heuristic evaluation in one very specific respect - it provides detailed quantitative information about usability. KLM is developed from Card, Moran & Newell's ***Model Human Processor***. It aims to provide a simplified model of human performance that is sufficient for predicting speed of interaction with a user interface. KLM assumes that the user already knows the sequence of operations that he or she is going to perform - that is, he or she is an ***expert user*** performing a ***routine task***. Card, Moran & Newell's ***GOMS*** model additionally takes into account the time required to plan more complex operations, but the use of GOMS for performance prediction is too complex to be presented here. The complexity of GOMS also means that it is of questionable utility compared to the relatively simple KLM (some well-known HCI researchers created a spoof satirising GOMS - you can see the GOMSerciser at: http://www.markroseman.com/goms.html).

The basic approach of KLM is to decompose the total task into ***unit operations***, using an established set of prediction components to estimate the time required to perform each operation, and add these estimated times in order to predict the overall task completion time.

Basic components

KLM has four basic time prediction components based on human performance:

- **K**: The time that it takes to press a ***key***. This is assumed to be constant, but depends on both the task and the typing skill of the user. A good typist takes 0.12 seconds on average to press keys, while an average typist takes 0.28 seconds. For difficult tasks, such as typing complex codes, 0.75 seconds is more typical, while a very inexperienced typist might take 1.2 seconds.

- **H**: The time that it takes to move your hands to the ***home*** position on a device (mouse or keyboard) = 0.40 seconds.

- **P**: The time that it takes to ***point*** with a mouse. This is predicted by ***Fitts' law,*** based on the size of the target and how far away it is. The result typically varies between 0.8 to 1.5 seconds, with an average of 1.1 seconds.

- **D**: The time that it takes to ***draw*** using a mouse. Card et. al. give a value based on a very primitive drawing algorithm that is probably not relevant to modern devices.

A further component describes the additional time taken while the user waits for the system to do something:

- **R**: The time that the system takes to ***respond*** to an action.

Finally, there is a component that estimates the time the user spends thinking before carrying out a unit operation:

- **M**: The time that it takes to ***mentally*** prepare for an action. This is estimated as a constant of 1.35 seconds.

Mental preparation

The main psychological subtlety in KLM is the question of when the user is expected to need mental preparation time. A set of rules define these circumstances for the purposes of estimation. The rules basically state that every operation must be preceded by mental preparation, but that no mental preparation is needed between two unit operations that form a ***chunk***. The definition of a chunk, however, is slightly ambiguous, as can be seen from the following rules.

Start by listing all of the operations that are required to complete the task, including points at which the system must respond before the user continues. Then apply the following rules:

Rule 0: Insert **M**s in front of all **K**s that are not part of a string (either text string or number string). Insert **M**s in front of all **P**s that select commands.

Rule 1: If the operator after an **M** is *fully anticipated* by the operator before it, the **M** can be deleted.

Rule 2: If a string of **K**s belong to a *cognitive unit*, such as a command name, the **M**s between them can be deleted.

Rule 3: Two **K**s that are both *terminators* (e.g. Return keys) do not need to be separated by an **M**.

Rule 4: Where a particular command string is always followed by a terminator, the terminator can be regarded as part of the command string, so no **M** is needed between them.

Applicability

The keystroke level model is a useful approach to analysing situations in which a user interface has a limited number of features, and these are used in a repetitive way. It provides detailed time and motion estimates that can be used to predict the improvements that would result from relatively minor changes to a user interface (or possibly to compare constrained and equivalent parts of alternative interfaces for a particular task). It is only really useful as a means for making comparative estimates - the absolute accuracy of the time estimates can vary quite widely according to user and task, and should probably be confirmed in practice by empirical measurements. Furthermore the chunking rules are rather ambiguous (e.g. the meaning of *fully anticipated*), and only apply to command-based systems. Equivalent chunking rules for phenomena such as dismissal of Windows dialogues would have to be established by further investigation.

A more complete discussion of GOMS and KLM is provided in the chapter by Bonnie John that appears in Carroll's course text. The above description is paraphrased (with additional critical comments) from the original paper describing the Keystroke Level Model:
S.K. Card, T.P. Moran and A. Newell, (1980). The keystroke-level model for user performance time with interactive systems. Communications of the ACM 23(7), 396-410. That paper is reprinted in Buxton & Baecker *Readings in Human Computer Interaction*: see the further reading list if you have an historical interest – but Bonnie John's chapter includes all you need to know.

Analysis Technique 3: Cognitive Walkthrough

The Cognitive Walkthrough (CW) method is very different to Keystroke Level Models. Where KLM can only analyse the performance of an expert user carrying out routine operations, CW assesses the usability of a system in situations where the user is not an expert, and may be attempting a task that he or she has never done before. The authors, Lewis and Polson, achieve this by applying their own theory of exploratory learning, called "CE+". Further details of the CE+ theory are not important - the CW method itself expresses quite clearly what are the assumptions and applicability of the theory.

Behaviour model

The model of a user carrying out a task through exploratory learning involves four basic phases:

1) The model describes how a notional user sets a ***goal*** to be accomplished with the system. A typical goal will be expressed in terms of the expected capabilities of the system, such as "check spelling of this document".

2) The model describes how the notional user searches the interface for currently available ***actions***. The availability of actions may be observable as the presence of menu items, of buttons, of available command-line inputs, etc.

3) The model describes how the notional user ***selects*** the action that seems likely to make progress toward the goal.

4) The model describes how the notional user ***performs*** the selected action and ***evaluates*** the system's feedback for evidence that progress is being made toward the current goal.

Evaluation procedure

The evaluation procedure is based on a manual ***simulation*** of a notional user iteratively carrying out the stages of the behavioural model. Note that users themselves are not involved – CW is a methodical approach to guessing the needs of real users, but without the difficulty and expense of recruiting actual sample users for observation. Before evaluation can start, the evaluators need to have access to the following information:

1) A general description of the ***type of users*** who would be expected to use the system, and the ***relevant knowledge*** that these users would be expected to have.

2) A description of one or more ***representative tasks*** to be used in the evaluation.

3) For each of the tasks, a list of the ***correct actions*** that should be performed in order to complete the task.

The evaluation is conducted by the interface designer, and by a ***group of peers***. This group includes a nominated ***scribe*** who records the results of the evaluation and a ***facilitator*** who is responsible for the smooth running of the evaluation process. The scribe and the facilitator are also active members of the evaluation group.

The group of evaluators move through each of the tasks, considering the user interface at each step. At each step, they examine the interface and tell a ***story*** about why the notional user would choose that action. These stories are then evaluated according to an information-processing model derived from the exploratory learning behavioural model:

1) consider what the notional user's current ***goal*** would be;

2) evaluate the ***accessibility*** of the correct control;

3) evaluate the quality of the *match* between the control's label and the goal; and

4) evaluate the *feedback* that would be provided to the notional user after the action.

Applicability

Cognitive walkthrough is widely considered to be based on a realistic model of system use, and one that is applicable to the current generation of WIMP / direct manipulation interfaces. It does assume that the evaluators are knowledgeable designers who are able to assess visibility, feedback and goal structures using relevant theories from cognitive psychology. It is more structured than heuristic evaluation, and is probably less likely to suffer from subjectivity as a result of its emphasis on the user (imagine a group of designer/evaluators arguing about the aesthetic quality of an interface that one of them has designed).

More information on cognitive walkthrough is available in a brief description presented at the ACM conference on Human Factors in Computing Systems in 1995:

```
http://doi.acm.org/10.1145/223355.223735
```

An alternative printed source is this chapter in a book on usability inspection (available in the CL library):

Wharton, C., Rieman, J., Lewis, C., and Polson, P. The cognitive walkthrough method: A practitioner's guide. In J. Nielsen and R. Mack (Eds.), Usability inspection methods. John Wiley & Sons, Inc., New York, NY, 1994.

Analysis Technique 4: Cognitive dimensions of notations

The cognitive dimensions of notations framework (CDs) is intended to provide a ***broad-brush*** approach to usability analysis. Its originator, Thomas Green, considered that earlier usability methods such as KLM and GOMS suffered a "death by detail" - the analysis results were at such a low level that designers could lose track of the best way to improve the interface. Green therefore set out to provide a tool that was directly usable by engineers, although reflecting his own psychological expertise.

The CDs are presented as a ***vocabulary*** for design discussion. Many of the dimensions reflect common usability factors that experienced designers might have noticed, but did not have a name for. Giving them a name allows designers to discuss these factors easily. Furthermore, CDs are based on the observation that there is no perfect user interface. Any user interface design reflects a set of design ***trade-offs*** that the designers have had to make. Giving designers a discussion vocabulary means that they can discuss the trade-offs that result from their design decisions. The nature of the trade-offs is reflected in the structure of the dimensions. It is not possible to create a design that has perfect characteristics in every dimensions - making improvements along one dimension often results in degradation along another.

An example dimension is called ***viscosity***, meaning resistance to change. In some notations, small conceptual changes can be very expensive to make. Imagine changing a variable from int to long in a large Java program. The programmer has to find every function to which that variable is passed, check the parameter declarations, check any temporary local variables where it is stored, check any calculations using the value, and so on. The idea of what the programmer needs to do is simple, but achieving it is hard. This is viscosity. There are programming languages that do not suffer from this problem, but they have other problems instead – trade-offs. This means that language designers must be able to recognise and discuss such problems when planning a new language. The word "viscosity" helps that discussion to happen.

Information artefacts

CDs are relevant to a wide range of situations including household appliances, telephones, and novel interaction devices as well as programming languages and other computer systems. These systems all provide a ***notation*** of some kind, and an ***environment*** for viewing and manipulating the notation. Usability is a function of both the notation and the environment.

Representative cognitive dimensions

The following list gives brief definitions of the main dimensions, and examples of the questions that can be considered in order to determine the effects that these dimensions will have on different user activities.

Premature commitment: constraints on the order of doing things.
> When you are working with the notation, can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first? If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?

Hidden dependencies: important links between entities are not visible.
> If the structure of the product means some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies are hidden? In what ways can it get worse when you are creating a particularly large description? Do these dependencies stay the same, or are there some actions that cause them to get frozen? If so, what are they?

Secondary notation: extra information in means other than formal syntax.

> Is it possible to make notes to yourself, or express information that is not really recognised as part of the notation? If it was printed on a piece of paper that you could annotate or scribble on, what would you write or draw? Do you ever add extra marks (or colours or format choices) to clarify, emphasise or repeat what is there already? If so, this may constitute a helper device with its own notation.

Viscosity: resistance to change.

> When you need to make changes to previous work, how easy is it to make the change? Why? Are there particular changes that are especially difficult to make? Which ones?

Visibility: ability to view components easily.

> How easy is it to see or find the various parts of the notation while it is being created or changed? Why? What kind of things are difficult to see or find? If you need to compare or combine different parts, can you see them at the same time? If not, why not?

Closeness of mapping: closeness of representation to domain.

> How closely related is the notation to the result that you are describing? Why? (Note that if this is a sub-device, the result may be part of another notation, not the end product). Which parts seem to be a particularly strange way of doing or describing something?

Consistency: similar semantics are expressed in similar syntactic forms.

> Where there are different parts of the notation that mean similar things, is the similarity clear from the way they appear? Are there places where some things ought to be similar, but the notation makes them different? What are they?

Diffuseness: verbosity of language.

> Does the notation a) let you say what you want reasonably briefly, or b) is it long-winded? Why? What sorts of things take more space to describe?

Error-proneness: the notation invites mistakes.

> Do some kinds of mistake seem particularly common or easy to make? Which ones? Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?

Hard mental operations: high demand on cognitive resources.

> What kind of things require the most mental effort with this notation? Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?

Progressive evaluation: work-to-date can be checked at any time.

> How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not? Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not? Can you try out partially-completed versions of the product? If not, why not?

Provisionality: degree of commitment to actions or marks.

> Is it possible to sketch things out when you are playing around with ideas, or when you aren't sure which way to proceed? What features of the notation help you to do this? What sort of things can you do when you don't want to be too precise about the exact result you are trying to get?

Role-expressiveness: the purpose of a component is readily inferred.

> When reading the notation, is it easy to tell what each part is for? Why? Are there some parts that are particularly difficult to interpret? Which ones? Are there parts that you really don't know what they mean, but you put them in just because it's always been that way? What are they?

Abstraction: types and availability of abstraction mechanisms.

> Does the system give you any way of defining new facilities or terms within the notation, so that you can extend it to describe new things or to express your ideas more clearly or succinctly? What are they? Does the system insist that you start by defining new terms before you can do anything else? What sort of things? These facilities are provided by an abstraction manager - a redefinition device. It will have its own notation and set of dimensions.

Sub-devices

Complex systems can include several specialised notations to help with a specific part of the job. Some of these might even seem to be separate from the rest of the system, for example when a user sticks a Post-It note on the computer screen, as a reminder of what to write in a word processor document. There are two kinds of these **sub-devices**.

● The Post-It note is an example of a **helper device**. When you make notes of telephone numbers on the back of an envelope, the complete system is the telephone plus the paper notes – if you didn't have some kind of helper device like the envelope, the telephone would be much less useful.

● A **redefinition device** changes the main notation in some way – such as defining a keyboard shortcut, a quick-dial code on a telephone, or a macro function. The redefinition device allows you to define these shortcuts, redefine them, delete them and so on.

Sub devices such as helper devices and redefinition devices often have their own notations that are separate from the main notation of the system, and an independent set of cognitive dimensions. The dimensions of these devices must be analysed independently.

Notational activities

When users interact with notations, there are a limited number of activities that they can engage in, when considered with respect to the way the notation might change. A CDs evaluation must consider which classes of activity will be the primary type of interaction for all representative system users. If the needs of different users have different relative priorities, those activities can be emphasised when design trade-offs are selected. The basic list of activities includes:

Search

> Finding information by navigating through the notational structure, using the facilities provided by the environment (e.g. finding a specific value in a spreadsheet). The notation is not changing at all, though the parts of it that the users sees will vary. Visibility and hidden dependencies can be important factors in search.

Incrementation

> Adding further information to a notation without altering the structure in any way (e.g. adding a new formula to a spreadsheet). If the structure will not change, then viscosity is not going to be very important.

Modification

Changing an existing notational structure, possibly without adding new content (e.g. changing a spreadsheet for use with a different problem).

Transcription

Copying content from one structure or notation to another notation (e.g. reading an equation out of a textbook, and converting it into a spreadsheet formula).

Exploratory design

Combining incrementation and modification, with the further characteristic that the desired end state is not known in advance (e.g. programming a spreadsheet on the fly or "hacking"). Viscosity can make this kind of activity far more difficult. This is why good languages for hacking may not be strictly typed, or make greater use of type inference, as maintaining type declarations causes greater viscosity. Loosely typed languages are more likely to suffer from hidden dependencies (a trade-off with viscosity), but this is not such a problem for exploratory design, where the programmer can often hold this information in his head during the relatively short development timescale.

Evaluation procedure

Cognitive dimensions are intended to be useful in a range of research and design contexts, and it is possible to apply them without a strict evaluation procedure. In a tutorial presented in 1998, Green and Blackwell suggested the following approach to system evaluation using CDs:

1) Identify the main *notation* of the system, describing the *medium* in which the marks of the notation are expressed, and the *environment* in which it is manipulated.

2) Identify *sub-devices*: helper devices and redefinition devices (some sub-devices may only become apparent once the analysis of the main notation is under way). Describe the notation used by each sub-device. Some systems also include other *layers* of notation where the system regenerates the same information in different notational forms. These must also be analysed separately, if the user ever interacts with them.

3) Consider each notation in terms of the list of *dimensions*, identifying any usability problems where the system characteristics on that dimension are inappropriate to the user activity (for example, high viscosity is inappropriate to exploratory design).

4) Where problems have been identified, consider *design manoeuvers* to adjust that dimension (design manoeuvres are described in more detail in other publications on cognitive dimensions). Many design manoeuvres introduce *trade-offs* that must be considered before finalising the change.

Applicability

Cognitive dimensions can be more useful as an early design tool than evaluation techniques are. They provide quite specific guidance regarding potential changes that can be made to the interface, as well as the likely consequences of those changes. They can also be used to analyse complex types of software, such as spreadsheets or programming languages, that cannot be analysed meaningfully using other evaluation techniques. At present there is no detailed evaluation procedure defined for use with CDs, but it is clear that they offer the potential to provide more information than other evaluation methods regarding the relative needs of different classes of user. They were originally developed in order to overcome sterile debates about the relative advantages of different programming languages, and this pragmatic orientation is highly valuable within a design team that must find alternatives to the search for an idealised solution to interface problems.

One of the chapters in the Carroll book gives a more extended description of Cognitive Dimensions, with examples and theoretical background. Further information on Cognitive Dimensions research can be found on-line in the Cognitive Dimensions archive, which is based in Cambridge.

```
http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/
```

Exercises

These exercises are intended as a basis for discussion during supervisions. Each of them requires several hours work, when done properly.

Interface Evaluation

1. Conduct a usability evaluation of the program that you use to read, send and archive email. Use either the cognitive walkthrough method or the cognitive dimensions of notations framework. Stop when you have identified five usability problems, and see if you can use the results of your analysis to suggest design changes that would fix these problems.

2. Use Keystroke Level Modeling to calculate the relative amounts of time that would be required to change some representative piece of text from "Sentence case" (capitalised like a sentence) to "Title Case" (capitalised like a title) for a concert poster that has been designed using a) your favourite word processing program, and for comparison, b) your favourite drawing program.

Interface Design

Sketch a basic user interface for an online application that will match students up with potential supervisors in the Computer Lab. The application should allow students to choose who will be in their tutorial groups, and it should allow changes to be made. See if you can anticipate problems with the interface, using either heuristic evaluation or cognitive dimensions of notations. Create a low-fidelity prototype of your interface using paper and pencil. Ask a friend to operate the prototype (while you provide system responses), and write a brief report describing the usability problems you discovered - compare these to the predictions you made.

Task Observation

Choose a situation in which you can watch people carrying out routine paper-based tasks, such as the queue at a building society, travel agent or post office, or an unobtrusive corner of a university department or college office. Spend five to ten minutes observing their activities and taking notes. Try not to get arrested. Describe the task that this person carries out, and identify a user profile suitable for use in a cognitive walkthrough evaluation of a new computer system (characteristics of the user, relevant knowledge etc.). Consider software packages that you are familiar with, and see if you can identify one that might be useful in this situation. What adaptation would be required?

Further resources

Recommended Text 1: Preece, Sharp & Rogers

Interaction Design: Beyond human-computer interaction. Jenny Preece, Helen Sharp & Yvonne Rogers, Wiley 2002

This is the second undergraduate HCI text written by this team – an unusual thing to happen. The amount of effort involved in writing a new textbook from scratch is so huge that very few people do it twice – as you might have noticed, most authors just issue revised editions every few years. Preece, Rogers and Sharp made the decision to rewrite their earlier successful textbook for three reasons: firstly, the new book is attractive, accessible, and clearly structured, to an extent that makes it an enjoyable read (in my opinion). Secondly, this book includes brief contributions from leading HCI practitioners in industry, which complements the academic material by giving an idea of how these ideas get applied in the professional world of software development and product design. Thirdly, this new book is completely up to date, in a field that has been changing rapidly over the last 10 years.

There has lately been an explosion in new research and usability analysis methods that come from outside the cognitive psychology tradition, and this book gives a broad review of them. These alternative methods are especially important in the design of social collaborative systems and "information appliances." Many older HCI textbooks emphasise the user interface as a controlled channel of interaction between the user and machine, with the keyboard, screen and mouse being the physical layer of that channel. That view of interaction has been amenable to cognitive descriptions, including known characteristics of human perception and motion control (as in the structure of this course). Systems that involve multiple users, or have no keyboard or screen require a broader repertoire of techniques for analysis. This is currently the only text that gives a good overview of these new developments, while also providing practical coverage of how traditional HCI techniques are used within professional contexts. It has been widely acclaimed for that reason.

Recommended Text 2: Carroll

HCI Models, Theories and Frameworks: Toward a multidisciplinary science. Edited by John. Carroll. Morgan Kaufmann 2003

Carroll is one of the first generation of HCI researchers, who was involved in several of the fundamental developments in the field, and he is still a very active and well-known researcher. In editing this book, he set out to create an advanced text that is suited to first year postgraduate students, by inviting leading researchers in the most important HCI research areas to summarise and review their specialist topics. Unlike many research collections, the resulting manuscript was tested in use on an actual university course through 2001/2002, and student feedback from that course was used to revise the final published version.

If the HCI curriculum in Cambridge spanned two years (as at many universities), Preece, Rogers and Sharp would be the part IB textbook, and this would be the part II textbook. However because the Cambridge HCI course is compressed into a single term, it is only possible to dip into the advanced HCI methods discussed in this book. There are several topics in the course that do extend into current research areas, and this book will be invaluable in understanding them fully. This book will also be an extremely valuable resource for any student considering further studies or research in HCI. As with Preece, Rogers and Sharp, reviews after its recent publication have agreed that it is the leading text of its kind. Of the two books, Preece, Rogers and Sharp is more relevant to understanding real commercial practice in HCI, and as a

resource for software engineers. Carroll is of more value in understanding the diverse range of theory that underlies HCI.

Other Books

William M. Newman, & Michael G. Lamming, Interactive System Design. Addison-Wesley 1995.

> This was the main textbook for the Cambridge HCI course until 2003, and there are still copies in the Computer Laboratory library. William Newman and Mik Lamming taught the first HCI course in Cambridge, while they were working as the Xerox Research Centre in Cambridge – once a preeminent center of HCI research in Europe, but sadly closed now. They wrote this book based on their experience of teaching in Cambridge, so it is well suited to the needs of Cambridge students. It has an emphasis on HCI as providing a set of engineering tools, the same attitude that still motivates the current course. The spirit of this book therefore remains in the lecture material, and in these lecture notes, but rather than reiterate this material in the course text, I have now chosen to complement the lecture material with two texts that extend it and introduce further topics.

Nathaniel S. Borenstein. Programming as if People Mattered: Friendly programs, software engineering and other noble delusions. Princeton University Press, 1991.

> Borenstein bases many of his observations on his personal experience of working on a large project: the Andrew Project developed by IBM and Carnegie Mellon University. The Andrew project pioneered many of the user interface facilities that are now central to the Internet, so he is able to discuss them from the perspective of a user interface designer who has seen the results of his work widely distributed and modified. His role as a manager gives him a similar perspective to that of Fred Brooks, author of the classic software engineering text *The Mythical Man-Month*. This book can be regarded as a user interface developer's version of *The Mythical Man-Month*.

William S. Buxton & Ronald M. Baecker (Eds.) (1987). Readings in Human Computer Interaction: A multidisciplinary approach. Morgan Kaufmann.

> This book has a similar ambition to Carroll's as can be seen in the title, but it mostly reprints classic research papers in HCI, rather than writing specially for a student audience. Many of the topics covered in the current course were first described in papers that can be found here. Examples include the Keystroke-Level Model, Mental Models, Direct Manipulation, and the history of the Xerox Star project.

Alan Cooper (1995). About Face: The Essentials of User Interface Design. IDG Books.

> This book is less research-oriented, but is based on very sound opinions and experience. The main advantage of this book is that it is more up to date, and gives much practical advice of direct relevance to the current generation of Windows software development, with examples and case studies.

Colin Robson (1994). Experiment, Design and Statistics in Psychology (3rd edition). Penguin.

> A basic text on experiment design and simple statistical analysis techniques that are suitable for use in HCI experiments. Many more sophisticated texts are available! Preece, Sharp & Rogers also give a brief introduction to this topic that may be sufficient.

Ericsson, K.A. & Simon, H.A. (1985). Protocol Analysis: verbal reports as data. MIT Press.

> A detailed text giving advice on the collection of verbal protocols, as well as a defense of the verbal protocol technique against criticisms of subjectivity or invalidity.

Internet Resources

HCI Bibliography

```
http://www.hcibib.org/
```
A searchable bibliography of the HCI research literature. This provides pretty much a single source for the professional HCI researcher. It is interesting to compare the novel search interface on this site to other online services. It was designed as a result of experiments conducted by John Pane during research for his PhD (and presented at the Psychology of Programming Interest Group) in which he identified and resolved common confusion that many non-programmers experience with Boolean set operators.

Jakob Nielsen's website

```
http://www.useit.com/
```
A respected HCI researcher who gained a high public profile while promoting Web usability for Sun. His advice on making usable web sites is particularly valuable and popular (the design of the site itself follows his advice, of course). Some of his claims, while good at promoting his consultancy business, should be read with a critical eye.

The US National Cancer Institute's web usability guidelines:

```
http://www.usability.gov/guidelines/
```

Microsoft's interface guidelines:

```
http://msdn.microsoft.com/library/,
```
search for "user interface guidelines"

Pruitt and Grudin's description of the use of Personas at Microsoft

```
http://research.microsoft.com/research/coet/Grudin/Personas/Pruitt-
Grudin.pdf
```

Cognitive Task Analysis using Interacting Cognitive Subsystems

```
http://www.shef.ac.uk/~pcljm/papers/ICS.NATO.pdf
```