Chapter 8

Design, prototyping and construction

8.1 Introduction

Design activities begin once a set of requirements has been established. Broadly speaking, there are two types of design: conceptual and physical. The former is concerned with developing a conceptual model that captures what the product will do and how it will behave, while the latter is concerned with details of the design such as screen and menu structures, icons, and graphics. The design emerges iteratively, through repeated design-evaluation-redesign cycles involving users.

For users to effectively evaluate the design of an interactive product, designers must produce an interactive version of their ideas. In the early stages of development, these interactive versions may be made of paper and cardboard, while as design progresses and ideas become more detailed, they may be polished pieces of software, metal, or plastic that resemble the final product. We have

called the activity concerned with building this interactive version prototyping and construction.

There are two distinct circumstances for design: one where you're starting from scratch and one where you're modifying an existing product. A lot of design comes from the latter, and it may be tempting to think that additional features can be added, or existing ones tweaked, without extensive investigation, prototyping or evaluation. It is true that if changes are not significant then the prototyping and evaluation activities can be scaled down, but they are still invaluable activities that should not be skipped.

In Chapter 7, we discussed some ways to identify user needs and establish requirements. In this chapter, we look at the activities involved in progressing a set of requirements through the cycles of prototyping to construction. We begin by explaining the role and techniques of prototyping and then explain how prototypes may be used in the design process. Tool support plays an important part in development, but tool support changes so rapidly in this area that we do not attempt to provide a catalog of current support. Instead, we discuss the kinds of tools that may be of help and categories of tools that have been suggested.

The main aims of this chapter are to:

- Describe prototyping and different types of prototyping activities.
- Enable you to produce a simple prototype.
- Enable you to produce a conceptual model for a system and justify your choices.
- Enable you to attempt some aspects of physical design.
- Explain the use of scenarios and prototypes in conceptual design.
- Discuss standards, guidelines, and rules available to help interaction designers.
- Discuss the range of tool support available for interaction design.

8.2 Prototyping and construction

It is often said that users can't tell you what they want, but when they see something and get to use it, they soon know what they don't want. Having collected information about work practices and views about what a system should and shouldn't do, we then need to try out our ideas by building prototypes and iterating through several versions. And the more iterations, the better the final product will be.

8.2.1 What *is a* prototype?

When you hear the term prototype, you may imagine something like a scale model of a building or a bridge, or maybe a piece of software that crashes every few minutes. But a prototype can also be a paper-based outline of a screen or set of screens, an electronic "picture," a video simulation of a task, a three-dimensional paper and cardboard **mockup** of a whole workstation, or a simple stack of **hyper**-linked screen shots, among other things.

In fact, a prototype can be anything from a paper-based storyboard through to a complex piece of software, and from a cardboard mockup to a molded or pressed piece of metal. A prototype allows stakeholders to interact with an envisioned product, to gain some experience of using it in a realistic setting, and to explore imagined uses.

For example, when the idea for the PalmPilot was being developed, Jeff Hawkin (founder of the company) carved up a piece of wood about the size and shape of the device he had imagined. He used to carry this piece of wood around with him and pretend to enter information into it, just to see what it would be like to own such a device (Bergman and Haitani, 2000). This is an example of a very simple (some might even say bizarre) prototype, but it served its purpose of simulating scenarios of use.

Ehn and Kyng (1991) report on the use of a cardboard box with the label "Desktop Laser Printer" as a mockup. It did not matter that, in their setup, the printer was not real. The important point was that the intended users, journalists and typographers, could experience and envision what it would be like to have one of these machines on their desks. This may seem a little extreme, but in 1982 when this was done, desktop laser printers were expensive items of equipment and were not a common sight around the office.

So a prototype is a limited representation of a design that allows users to interact with it and to explore its suitability.

8.2.2 Why prototype?

Prototypes are a useful aid when discussing ideas with stakeholders; they are a communication device among team members, and are an effective way to test out ideas for yourself. The activity of building prototypes encourages reflection in design, as described by Schon (1983) and as recognized by designers from many disciplines as an important aspect of the design process. Liddle (1996), talking about software design, recommends that prototyping should always precede any writing of code.

Prototypes answer questions and support designers in choosing between alternatives. Hence, they serve a variety of purposes: for example, to test out the technical feasibility of an idea, to clarify some vague requirements, to do some user testing and evaluation, or to check that a certain design direction is compatible with the rest of the system development. Which of these is your purpose will influence the kind of prototype you build. So, for example, if you are trying to clarify how users might perform a set of tasks and whether your proposed device would support them in this, you might produce a paper-based mockup. Figure 8.1 shows a paper-based prototype of the design for a handheld device to help an autistic child communicate. This prototype shows the intended functions and buttons, their positioning and labeling, and the overall shape of the device, but none of the buttons actually work. This kind of prototype is sufficient to investigate scenarios of use and to decide, for example, whether the buttons are appropriate and the functions sufficient, but not to test whether the speech is loud enough or the response fast enough.

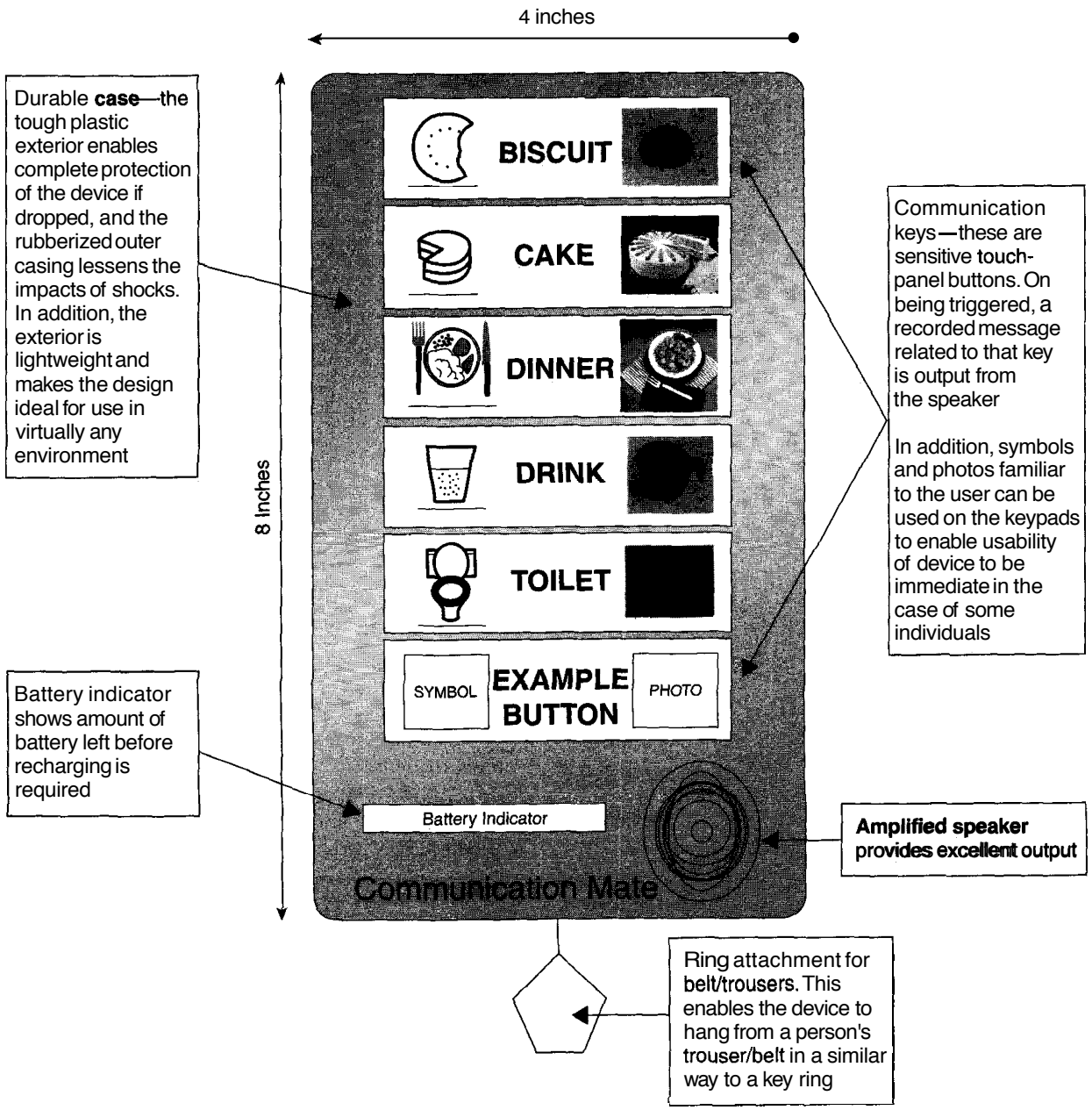


Figure 8.1 A paper-based ***prototype*** of a handheld device to support an autistic child.

Heather Martin and Bill Gaver (2000) describe a different kind of prototyping with a different purpose. When prototyping audiophotography products, they used a variety of different techniques including video scenarios similar to the scenarios we introduced in Chapter **7,** but filmed rather than written. At each stage, the prototypes were minimally specified, deliberately leaving some aspects vague so as to stimulate further ideas and discussion.

8.2.3 Low-fidelity prototyping

A low-fidelity prototype is one that does not look very much like the final product. For example, it uses materials that are very different from the intended final version, such as paper and cardboard rather than electronic screens and metal. The lump of wood used to prototype the Palm Pilot described above is a low-fidelity prototype, as is the cardboard-box laser printer.

Low-fidelity prototypes are useful because they tend to be simple, cheap, and quick to produce. This also means that they are simple, cheap, and quick to modify so they support the exploration of alternative designs and ideas. This is particularly important in early stages of development, during conceptual design for example, because prototypes that are used for exploring ideas should be flexible and encourage rather than discourage exploration and modification. Low-fidelity prototypes are never intended to be kept and integrated into the final product. They are for exploration only.

Storyboarding Storyboarding is one example of low-fidelity prototyping that is often used in conjunction with scenarios, as described in Chapter 7. A storyboard consists of a series of sketches showing how a user might progress through a task using the device being developed. It can be a series of sketched screens for a GUI-based software system, or a series of scene sketches showing how a user can perform a task using the device. When used in conjunction with a scenario, the storyboard brings more detail to the written scenario and offers stakeholders a chance to role-play with the prototype, interacting with it by stepping through the scenario. The example storyboard shown in Figure 8.2 (Hartfield and Winograd,

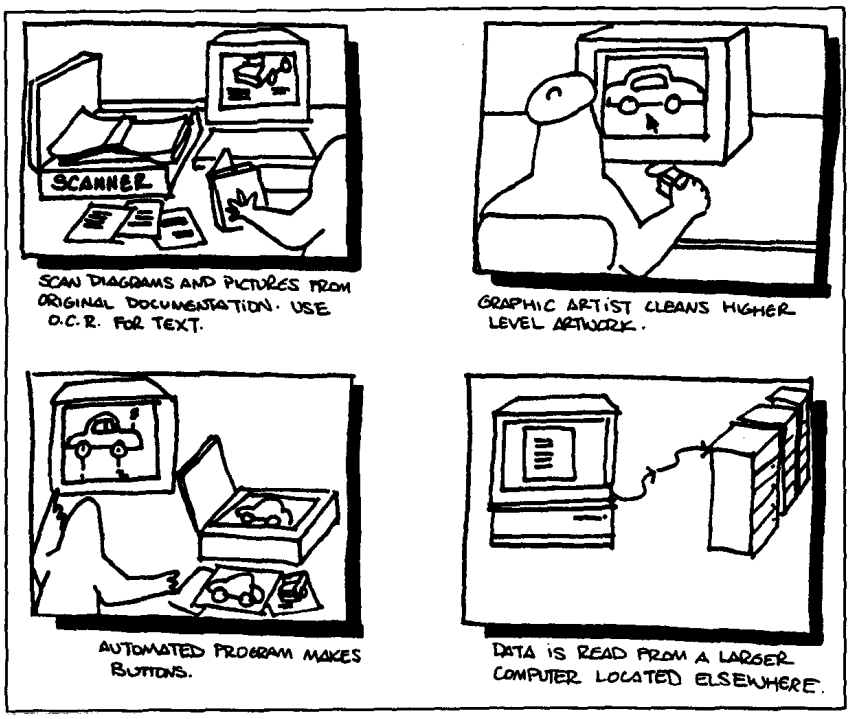


Figure 8.2 An example storyboard.

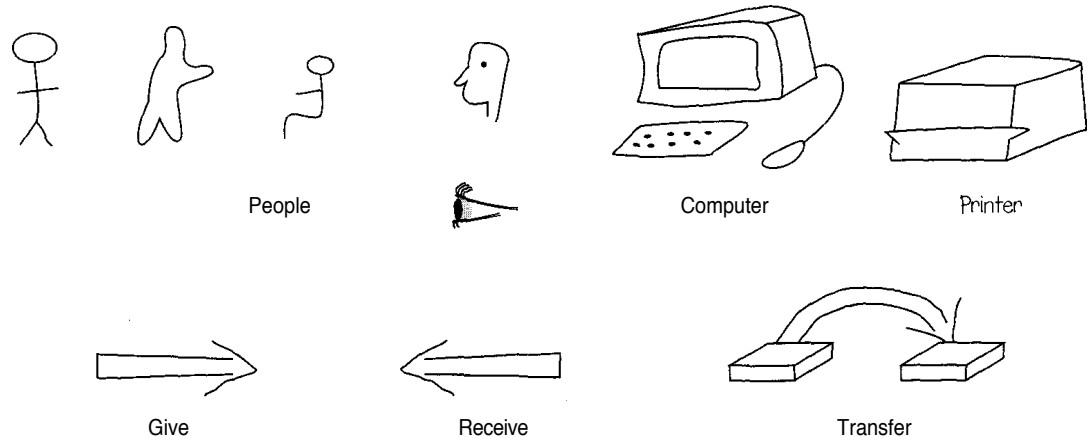


Figure 8.3 Some simple sketches for low-fidelity prototyping.

1996) depicts a person using a new system for digitizing images. This example doesn't show detailed drawings of the screens involved, but it describes the steps a user might go through in order to use the system.

Sketching Low-fidelity prototyping often relies on sketching, and many people find it difficult to engage in this activity because they are inhibited about the quality of their drawing. Verplank (1989) suggests that you can teach yourself to get over this inhibition. He suggests that you should devise your own symbols and icons for elements you might want to sketch, and practice using them. They don't have to be anything more than simple boxes, stick figures, and stars. Elements you might require in a storyboard sketch, for example, include "things" such as people, parts of a computer, desks, books, etc., and actions such as give, find, transfer, and write. If you are sketching an interface design, then you might need to draw various icons, dialog boxes, and so on. Some simple examples are shown in Figure **8.3.** Try copying these and using them. The next activity requires other sketching symbols, but they can still be drawn quite simply.

ACTIVITY 8.1 Produce a storyboard that depicts how to fill a car with gas (petrol).

Comment Our attempt is shown in Figure 8.4.

Prototyping with Index Cards Using index cards (small pieces of cardboard about **3** × 5 inches) is a successful and simple way to prototype an interaction, and is used quite commonly when developing websites. Each card represents one screen or one element of a task. In user evaluations, the user can step through the cards, pretending to perform the task while interacting with the cards. **A** more detailed example of this kind of prototyping is given in Section 8.3.4.

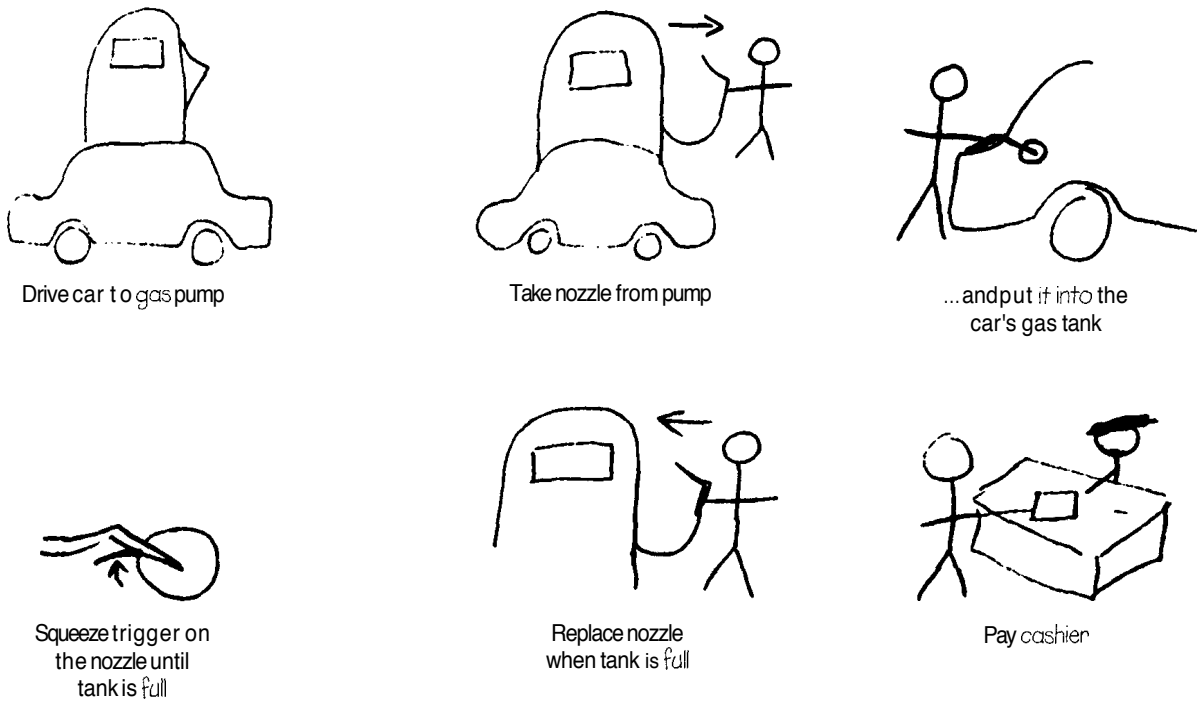


Figure 8.4 A storyboard depicting how to fill a car with gas.

Wizard **of** Oz Another low-fidelity prototyping method called Wizard of Oz assumes that you have a software-based prototype. In this technique, the user sits at a computer screen and interacts with the software as though interacting with the product. In fact, however, the computer is connected to another machine where a human operator sits and simulates the software's response to the user. The method takes its name from the classic story of the little girl who is swept away in a storm and finds herself in the Land of Oz (Baum and Denslow, 1900).

8.2.4 High-fidelity prototyping

High-fidelity prototyping uses materials that you would expect to be in the final product and produces a prototype that looks much more like the final thing. For example, a prototype of a software system developed in Visual Basic is higher fidelity than a paper-based mockup; a molded piece of plastic with a dummy keyboard is a higher-fidelity prototype of the PalmPilot than the lump of wood.

If you are to build a prototype in software, then clearly you need a software tool to support this. Common prototyping tools include Macromedia Director, Visual Basic, and Smalltalk. These are also full-fledged development environments, so they are powerful tools, but building prototypes using them can also be very straightforward.

Table 8.1 Relative effectiveness of low- vs. high-fidelity prototypes (Rudd et al., 1996)

Type	Advantages	Disadvantages
Low-fidelity prototype	• Lower development cost. • Evaluate multiple design concepts. • Useful communication device. • Address screen layout issues. • Useful for identifying market requirements. • Proof-of-concept.	• Limited error checking. • Poor detailed specification to code to. • Facilitator-driven. • Limited utility after requirements established. • Limited usefulness for usability tests. • Navigational and flow limitations.
High-fidelity prototype	• Complete functionality. • Fully interactive. • User-driven. • Clearly defines navigational scheme. • Use for exploration and test. • Look and feel of final product. • Serves as a living specification. • Marketing and sales tool.	• More expensive to develop. • Time-consuming to create. • Inefficient for proof-of-concept designs. • Not effective for requirements gathering.

Marc Rettig (1994) argues that more projects should use low-fidelity prototyping because of the inherent problems with high-fidelity prototyping. He identifies these problems as:

- They take too long to build.
- Reviewers and testers tend to comment on superficial aspects rather than content.
- Developers are reluctant to change something they have crafted for hours.
- A software prototype can set expectations too high.
- Just one bug in a high-fidelity prototype can bring the testing to a halt.

High-fidelity prototyping is useful for selling ideas to people and for testing out technical issues. However, the use of paper prototyping and other ideas should be actively encouraged for exploring issues of content and structure. Further advantages and disadvantages of the two types of prototyping are listed in Table 8.1.

8.2.5 Compromises in protoiyping

By their very nature, prototypes involve compromises: the intention is to produce something quickly to test an aspect of the product. The kind of questions or choices

BOX 8.1 Prototyping Cultures *(Schrage, 1996)*

"The culture of an organization has a strong influence on the quality of the innovations that the organization can produce." (Schrage, 1996, p. 193)

This observation is drawn mainly from product-related organizations, but also applies to software development. There are primarily two kinds of organizational culture for innovation: the specification culture and the prototyping culture. In the former, new products and development are driven by written specifications, i.e., by a collection of documented requirements. In the latter, understanding requirements and developing the new product are driven by prototyping. Large companies such as IBM or AT&T that have to gather and coordinate a large amount of information tend to be specification-driven, while smaller entrepreneurial companies tend to be prototype-driven. Both approaches have potential disadvantages. A carefully prepared specification may prove completely infeasible once prototyping begins. Similarly, a wonderful prototype may prove to be too expensive to produce on a large scale.

The medium used for developing the prototype affects the process itself. Schrage puts forward the example of General Motors, which used to produce clay prototypes of new cars and then try to capture these in CAD tools. On the other hand, Toyota designs its cars using CAD tools first and produces a clay prototype once the design has stabilized. The medium used also determines in part the questions that a prototype can answer. As a simple example, a horizontal software prototype will not be able to answer questions about the detailed operation of a function since it is not designed to model that level of detail.

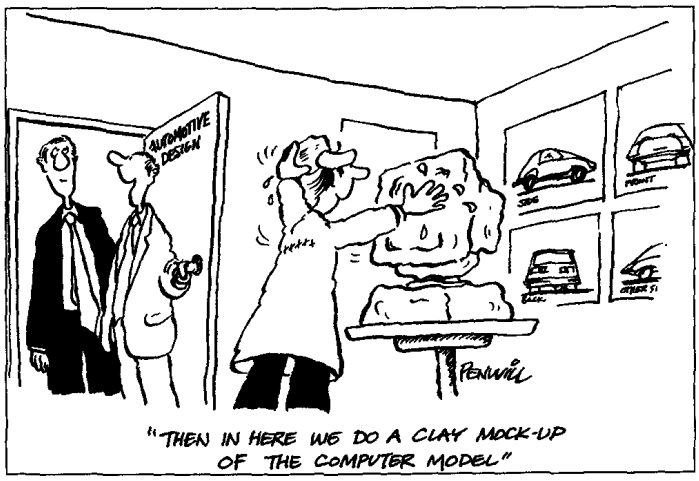
The speed of prototype development and the time between prototyping iterations is often a product of organizational culture and tradition. Some companies have a set number of prototypes embodied in their development method and use this number irrespective of the technical needs of any particular product. Generally speaking, the more prototyping cycles there are, the more polished the final product will be.

The corporate prototyping culture is most starkly revealed by who is involved in prototyping and when. For example, who owns the prototype? Is there a special prototyping department? Who gets to see and evaluate the prototype? Sometimes designers are happy to show emerging prototypes to their peers but not to managers, for fear of being misunderstood and either having the project cancelled or finding an order to ship the prototype when it is not ready. Prototype demonstrations to senior managers often happen too late in the development cycle to have any real impact because of these fears.

David Kelley (Schrage, p.195) claims that organizations wanting to be innovative need to move to a prototype-driven culture. Schrage sees that there are two cultural aspects to this shift. First, scheduled prototyping cycles that force designers to build many prototypes are more likely to lead to a prototype-driven culture than allowing designers to produce *ad hoc* prototypes when they think it appropriate. Second, rather than innovative teams being needed for innovative prototypes, it is now recognized that innovative prototypes lead to innovating teams! This can be especially significant when the teams are cross-functional, i.e., multidisciplinary.

that any one prototype allows the designer to answer is therefore limited, and the prototype must be designed and built with the key issues in mind. In low-fidelity prototyping, it is fairly clear that compromises have been made. For example, with a paper-based prototype an obvious compromise is that the device doesn't actually work! For software-based prototyping, some of the compromises will still be fairly clear; for example, the response speed may be slow, or the exact icons may be sketchy, or only a limited amount of functionality may be available.

Two common compromises that often must be traded against each other are breadth of functionality provided versus depth. These two kinds of prototyping



are called horizontal prototyping (providing a wide range of functions but with little detail) and vertical prototyping (providing a lot of detail for only a few functions).

Other compromises won't be obvious to a user of the system. For example, the internal structure of the system may not have been carefully designed, and the prototype may contain "spaghetti code" or may be badly partitioned. One of the dangers of producing running prototypes, i.e., ones that users can interact with automatically, is that they may believe that the prototype is the system. The danger for developers is that it may lead them to consider fewer alternatives because they have found one that works and that the users like. However, the compromises made in order to produce the prototype must not be ignored, particularly the ones that are less obvious from the outside. We still must produce a good-quality system and good engineering principles must be adhered to.

8.2.6 Construction: from design to implementation

When the design has been around the iteration cycle enough times to feel confident that it fits requirements, everything that has been learned through the iterated steps of prototyping and evaluation must be integrated to produce the final product.

Although prototypes will have undergone extensive user evaluation, they will not necessarily have been subjected to rigorous quality testing for other characteristics such as robustness and error-free operation. Constructing a product to be used by thousands or millions of people running on various platforms and under a wide range of circumstances requires a different testing regime than producing a quick prototype to answer specific questions.

The dilemma box below discusses two different development philosophies. One approach, called evolutionary prototyping, involves evolving a prototype into the final product. An alternative approach, called throwaway prototyping, uses the prototypes as stepping stones towards the final design. In this case, the

DILEMMA Prototyping to Throw Away

Low-fidelity prototypes are never intended to be kept and integrated into the final product. But when building a software-based system, developers can choose to do one of two things: either build a prototype with the intention of throwing it away after it has fulfilled its immediate purpose, or build a prototype with the intention of evolving it into the final product.

Above, we talked about the compromises made when producing a prototype, and we commented that the "invisible" compromises, concerned with the structure of the underlying software must not be ignored. However, when a project team is under pressure to produce the final product and a complex prototype exists that fulfills many of the requirements, or maybe a set of vertical prototypes exists that together fulfill the requirements, it can become very tempting to pull them together and issue the result as the final product. After all, many hours of development have probably gone into developing the prototypes, and evaluation with the client has gone well, so isn't it a waste to throw it all away? Basing the final product on prototypes in this way will simply store up testing and maintenance problems for later on: in short, this is likely to compromise the quality of the product.

Evolving the final prototype into the final product through a defined process of evolutionary prototyping can lead to a robust final product, but this must be clearly planned and designed for from the beginning. Building directly on prototypes that have been used to answer specific questions through the development process will not yield a robust product. As Constantine and Lockwood (1999) observe, "Software is the only engineering field that throws together prototypes and then attempts to sell them as delivered goods".

On the other hand, if your device is an innovation, then being first to market with a "good enough" product may be more important for securing your market position than having a very high-quality product that reaches the market two months after your competitors'.

prototypes are thrown away and the final product is built from scratch. If an evolutionary prototyping approach is to be taken, the prototypes should be subjected to rigorous testing along the way; for throw-away prototyping such testing is not necessary.

8.3 Conceptual design: moving from requirements to first design

Conceptual design is concerned with transforming the user requirements and needs into a conceptual model. Conceptual models were introduced in Chapter 2, and here we provide more detail and discuss how to go about developing one. We defined conceptual model as "a description of the proposed system in terms of a set of integrated ideas and concepts about what it should do, behave, and look like, that will be understandable by the users in the manner intended." The basis for designing this model is the set of user tasks the product will support. There is no easy transformation to apply to a set of requirements data that will produce "the best" or even a "good enough" conceptual model. Steeping yourself in the data and trying to empathize with the users while considering the issues raised in this section is one of the best ways to proceed. From the requirements and this experience, a picture of what you want the users' experience to be when using the new product will emerge.

Beyer and Holtzblatt (1998), in their method ***Contextual Design*** discussed in Chapter 9, recommend holding review meetings within the team to get different peoples' perspectives on the data and what they observed. This helps to deepen understanding and to expose the whole team to different aspects. Ideas will emerge as this extended understanding of the requirements is established, and these can be tested against other data and scenarios, discussed with other design team members and prototyped for testing with users. Other ways to understand the users' experience are described in Box 8.2.

Ideas for a conceptual model may emerge during data gathering, but remember what Suzanne Robertson said in her interview at the end of Chapter **7:** you must separate the real requirements from solution ideas.

Key guiding principles of conceptual design are:

- Keep an open mind but never forget the users and their context.
- Discuss ideas with other stakeholders as much as possible.
- Use low-fidelity prototyping to get rapid feedback.
- Iterate, iterate, and iterate. Remember Fudd's first law of creativity: "To get a good idea, get lots of ideas" (Rettig, 1994).

Considering alternatives and repeatedly thinking about different perspectives helps to expand the solution space and can help prompt insights. Prototyping (introduced in Section 8.2) and scenarios (introduced in Chapter **7**) are two techniques to help you explore ideas and make design decisions. But before explaining how these can help, we need to explore in more detail how to go about envisioning the product.

8.3.1 Three perspectives for developing a conceptual model

Chapter 2 introduced three ways of thinking about a conceptual model: Which interaction mode would best support the users' activities? Is there a suitable interface metaphor to help users understand the product? Which interaction paradigm will the product follow? In this section, we discuss each of these in more detail. In all the discussions that follow, we are not suggesting that one way of approaching a conceptual design is right for one situation and wrong for another; they all provide different ways of thinking about the product and hence aid in generating alternatives.

Which interaction mode? Which interaction mode is most suitable for the product depends on the activities the user will engage in while using it. This information is identified through the requirements activity. The interaction mode refers to how the user invokes actions when interacting with the device. In Chapter 2 we introduced two different types of interaction mode: those based on activities and those based on objects. For those based on activities, we introduced four general styles: instructing, conversing, manipulating and navigating, and exploring and browsing. Which is best suited to your current design depends on the application domain and the kind of system being developed. For example, a computer game is most likely to suit a manipulating and navigating style, while a drawing package has aspects of instructing and conversing.

BOX 8.2 How to Really Understand the Users' Experience

Some design teams go to great lengths to ensure that they come to empathize with, not just understand, the users' experience. We know from learning things ourselves that "learning by doing" is more effective than being told something or just seeing something. Buchenau and Suri (2000) describe an approach they call experience prototyping that is intended to give designers some of the insight into a user's experience that comes only from first-hand knowledge. For example, they describe a team designing a chest-implanted automatic defibrillator. A defibrillator is used with victims of cardiac arrest when their heart muscle goes into a chaotic arrythmia and fails to pump blood, a state called fibrillation. A defibrillator delivers an electric shock to the heart, often through paddle electrodes applied externally through the chest wall; an implanted defibrillator does this through leads that connect directly to the heart muscle. In either case, it's a big electric shock intended to restore the heart muscle to its regular rhythm that can be powerful enough to knock people off their feet.

This kind of event is completely outside most people's experience, and so it is difficult really to understand what the user's experience is likely to be for this kind of device. You can't fit a prototype pacemaker to each member of the design team and simulate fibrillation in them! This makes it difficult for designers to gain the insight they need. However, you can simulate some critical aspects of the experience, one of which is the random occurrence of a defibrillating shock. To achieve this, each team member was given a pager to take home over the weekend (elements of the pack are shown in Figure 8.5). The pager message simulated the occurrence of a defibrillating shock. Messages were sent at random, and team members were asked to record where they were, who they were with, what they were doing, and what they thought and felt knowing that this represented a shock. Experiences were shared the following week, and example insights ranged from anxiety around everyday happenings such as holding a child and operating power tools, to being in social situations at a loss how to communicate to onlookers what was happening. This first-hand experience brought new insights to the design effort.

Another instance in which designers tried hard to come to terms with the user experience is the Third Age suit, developed at ICE, Loughborough University (see Figure 8.6). This suit was designed so that car designers could experience what it

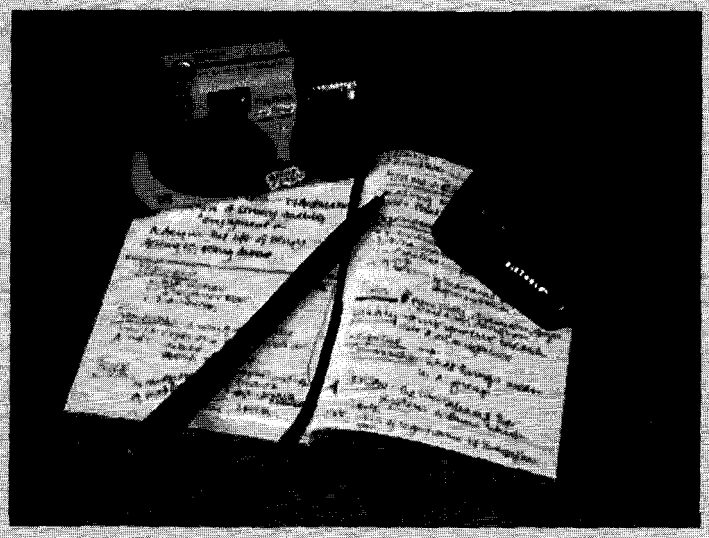


Figure 8.5 The patient kit for experience prototyping.

might be like to be in an older body. The suit restricts movement in the neck, arms, legs, and ankles in a way that simulates the mobility problems typically experienced by someone over 55 years of age. For example, when operating the foot pedals in a car, many "third agers" (as they are called), lack the flexibility in their ankles to be able to rest their heel on the floor and operate the pedals by flexing their ankle. Thus they have to lift their whole foot up and push it down each time they operate the pedal, which puts much more stress on their leg muscles.

Figure 8.6 The Third Age suit: (a) riding a bike and (b) using a mobile phone.

Most conceptual models will be a combination of modes, and it is necessary to associate different parts of the interaction with different modes. For example, consider the shared calendar example introduced in Chapter 7. One of the user tasks is finding out what is happening on a particular day. In this instance, instructing is an appropriate mode of interaction. No dialog is necessary for the system to show the required information. On the other hand, the user task of trying to arrange a meeting among a set of people may be conducted more like a conversation. We can imagine that the user begins by selecting the people for the meeting and setting some constraints on the arrangements such as time limit, urgency, length of meeting, etc. Then the system might respond with a set of possible times and dates for the user to select. This is much more like a conversation. (You may like to refer back to the scenario of this task in Chapter 7 and consider how well it matches this interaction mode.) For the task of planning, the user is likely to want to scan through pages and browse the days.

ACTIVITY 8.2 Consider the library catalog system introduced in Chapter 7. Identify tasks associated with this product that would be best supported by each of the interaction modes instructing, conversing, manipulating and navigating, and exploring and browsing.

Comment Here are some suggestions. You may have identified others:

(a) Instructing: the user wants to see details of a particular book, such as publisher and location.

(b) Conversing: the user wants to identify a book on a particular topic but doesn't know exactly what is required.

(c) Manipulating and navigating: the library books could be represented as icons that could be interrogated for information or manipulated to represent the book being reserved or borrowed.

(d) Exploring and browsing: the user is looking for interesting books, with no particular topic or author in mind.

Models based on objects provide a different perspective since they are structured around real-world objects. For example, the shared calendar system can be thought of as an electronic version of a paper calendar, which is a book kept by each person on their desk or in their bag. Alternatively, it could be thought of as a planner, a large flat piece of paper that is often pinned up on the wall in offices and is far more public. The choice of which objects to choose as a basis for the conceptual model is related to the choice of interface metaphor, which we consider below.

Mayhew (1999) identifies a similar distinction between conceptual models: process-oriented or product-oriented. The former kind of model best fits "an application in which there are no clearly identifiable primary work products. In these applications the main point is to support some work process." Examples of this might be software to control a chemical processing plant, a financial management package, or a customer care call-center. On the other hand, a product-oriented model "will best fit an application in which there are clear, identifiable work products that users individually create, modify and maintain." Examples of this are Microsoft products such as Excel, Powerpoint, Word, etc. More information about these kinds of conceptual model is given in Box 8.3.

Is there a suitable interface *metaphor*? Interface metaphors are another way to think about conceptual models. They are intended to combine familiar knowledge with new knowledge in a way that will help the user understand the system. Choosing suitable metaphors and combining new and familiar concepts requires a careful balance and is based on a sound understanding of the users and their context. For example, consider an educational system to teach six-year-olds mathematics. You could use the metaphor of a classroom with a teacher standing at the blackboard. But if you consider the users of the system and what is likely to engage them, you will be more likely to choose a metaphor that reminds the children of something they enjoy, such as a ball game, the circus, a playroom, etc.

Erickson (1990) suggests a three-step process for choosing a good interface metaphor. The first step is to understand what the system will do. Identifying functional requirements was discussed in Chapter 7. Developing partial conceptual models and trying them out may be part of the process. The second step is to understand which bits of the system are likely to cause users problems. Another way of looking at this is to identify which tasks or subtasks cause problems, are complicated, or are critical. A metaphor is only a partial mapping between the software and the real thing upon which the metaphor is based. Understanding areas in which users are likely to have difficulties means that the metaphor can be chosen to support those aspects. The third step is to generate metaphors. Looking for metaphors in the users' description of the tasks is a good starting point. Also, any

BOX 8.3 Process-Oriented versus Product-Oriented Conceptual Models *(Mayhew, 1999)*

Mayhew (1999) characterizes conceptual models in terms of their focus on products or on process. This is similar to our characterization of models that focus on objects or ones that focus on activities.

The difference between these two kinds of conceptual model is the drivers for the design activity. For a product-oriented system, the main products and the tools needed to create them form the main structure of the application. For a process-oriented application, it is the list of process steps that forms the system's basis. Mayhew suggests the following issues must be addressed during conceptual design, whether the application is primarily product-oriented or process-oriented:

- Products or processes must be clearly identified. For example, what documents are to be generated and what other tools are required to produce them? In a process-oriented model, what processes are to be supported?
- A set of presentation rules must be designed. For example, urgent tasks must be placed on the desktop, while less urgent tasks may be accessible through the menu bar. If designing for a GUI, design rules and guidelines come with the particular platform (see Box 8.5 below).
- Design a set of rules for how windows will be used.
- Identify how major information and functionality will be divided across displays.
- Define and design major navigational pathways. This will draw on the task analysis earlier, and leads to a structure for the tasks. Don't over-constrain users, make navigation easy, and provide facilities so that they always know where they are.
- Document alternative conceptual design models in sketches and explanatory notes.

An example conceptual model based on this approach is shown in Figure 8.7. This is a process-based model, and so it is structured around the processes and subprocesses the system is to support.

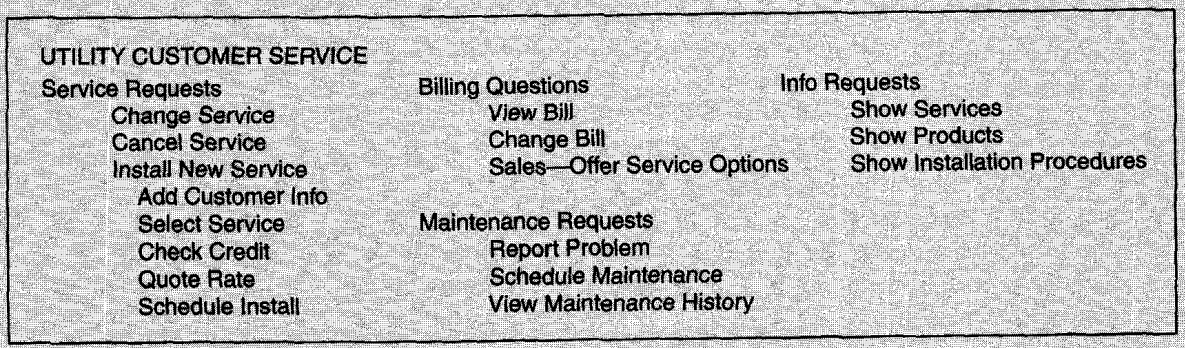


Figure 8.7 An example of a process-oriented conceptual model.

metaphors used in the application domain with which the users may be familiar may be suitable.

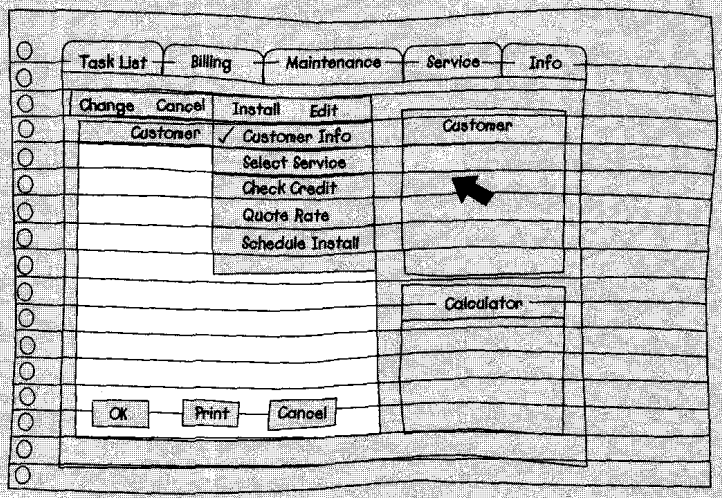
When suitable metaphors have been generated, they need to be evaluated. Again, Erickson (1990) suggests five questions to ask.

1. How much structure does the metaphor provide? A good metaphor will require structure, and preferrably familiar structure.

Here is a possible design presenting these work processes through a process-oriented Conceptual Model:

Conceptual Model—Utility Customer Service

Application Window



The presentation rules followed in this design are as follows (this design is not complete; it simply provides some examples of components of a process-oriented model).

The overall application is represented as a tab metaphor. *Highest-level processes* (e.g., billing questions, maintenance requests) are represented by tabs, and each tab represents a work space for that process. The tabbed work space includes a main window where that process is carried out, plus two "common windows," where tools common across all highest-level processes are maintained.

Second-level subprocesses are represented by selections in the menu bar within each tab, and *third-level subprocesses* by selections in pull-downs from the menu bar.

Structured subprocesses are controlled through dimming of subprocesses in pull-downs (until earlier subprocesses are completed, later subprocesses are dimmed out and unselectable).

Completed subprocesses are designated with a check mark.

Common activities available across highest-level processes (i.e., Customer, Calculator) are presented as separate, dedicated windows within the tabbed work spaces.

All windows are dialog boxes—that is, they cannot be minimized. They are all unresizable and unscrollable, but are movable and modeless. The main dialog box represents subprocesses, and it changes contents as the user moves through the subprocesses in sequence to complete a given subprocess and process.

Different windows have different background colors. Note that the active tabbed workspace itself is dark gray, the main subprocess dialog box is white, and all common dialog boxes are light gray.

Figure 8.7 (*continued*).

2. How much of the metaphor is relevant to the problem? One of the difficulties of using metaphors is that users may think they understand more than they do and start applying inappropriate elements of the metaphor to the system, leading to confusion or false expectations.
3. Is the interface metaphor easy to represent? **A** good metaphor will be associated with particular visual and audio elements, as well as words.

4. Will your audience understand the metaphor?
5. How extensible is the metaphor? Does it have extra aspects that may be useful later on?

In the calendar system, one obvious metaphor we could use is the individual's paper-based calendar. This is familiar to everyone, and we could combine that familiarity with facilities suitable for an electronic document such as hyperlinks and searching. Having thought of this metaphor, we need to apply the five questions listed above.

1. Does it supply structure? Yes, it supplies structure based on the familiar paper-based calendar. However, it does not supply structure for the notion of sharing information, i.e., other people looking in the calendar, because of two issues: first, an individual's calendar is very personal, and second, even if there is a paper-based calendar for a set of people, it can be closed and the information hidden from casual observers.
2. How much of the metaphor is relevant i.e., how many properties of the paper-based calendar are applicable to the electronic version? Well, in the electronic version it isn't appropriate to think of physically turning pages, but then a facility for looking at one "page" after another is required. The individual's calendar can be carried around from place to place. Whether or not we want to encourage that aspect of the metaphor depends on the kind of interaction paradigm we might consider. Finally, this is a shared calendar, and normally our personal calendars are not shared.
3. Is the metaphor easy to represent? Yes.
4. Will your audience understand the metaphor? Yes.
5. How extensible is the metaphor? The functionality of a paper-based calendar is fairly limited. However, it is also a book, and we could borrow facilities from electronic books (which are also familiar objects to most of our audience), so yes, it can be extended.

ACTIVITY 8.3 Another possible interface metaphor for the shared calendar system is the wall planner. Ask the five questions above of this metaphor.

Comment

(a) Does it supply structure? Yes, it supplies structure based on the wall-planner. This metaphor embodies the notion of public access more than the paper-based calendar. In particular, the wall planner is never "closed" to those who are near it.

(b) How much of the metaphor is relevant? Most of this metaphor is relevant. Individuals don't walk around with the wall planner, though, so the answer depends on how the calendar is to be used.

(c) Is the metaphor easy to represent? Yes, it could be represented as a spreadsheet.

(d) Will your audience understand the metaphor? Yes.

(e) How extensible is the metaphor? The functionality of a wall planner is also fairly limited. There are no obvious ways in which to extend the metaphor to help with this application.

Which interaction paradigm? Interaction paradigms are design philosophies that help you think about the product being developed. Interaction paradigms include the now traditional desktop paradigm, with WIMP interface (windows, icons, menus and pointers), ubiquitous computing, pervasive computing, wearable computing, tangible bits, attentive environments, and the Workaday World. Thinking about the user tasks with these different paradigms in mind can help provide insight both to choose the interaction paradigm and to inspire a different perspective on the problem.

Thinking about environmental requirements is particularly relevant when considering interaction paradigms. For example, consider the shared calendar in the context of the following paradigms:

- ***Ubiquitous computing.*** Combining some of our earlier discussions, we could perhaps imagine the shared calendar as being like a planner on the wall, but in an electronic form with which people could interact.

  Pervasive computing. Carrying around our own copy of the shared calendar builds directly upon current expectations and experience of personal calendars. We can imagine a system that allows individuals to keep a copy of the system on their own palmtop computers or PDAs, while also being linked to a central server somewhere that allows access to other information that is shared.
- ***Wearable computing.*** Imagine having an earring or a tie pin telling you that you have an appointment in an hour's time at a client's office and that you need to book a taxi? Or maybe asking you whether it is all right to book a meeting with your colleague on a particular date. What other possibilities can this model conjure up?

ACTIVITY 8.4 Consider the library catalog system and think about each of the paradigms listed above. Choose two of them and suggest different kinds of interaction that these paradigms imply.

Comment We had the following thoughts, but you may have others. The library catalog is likely to be used only in certain places, such as the library or perhaps in an office. The idea of wearable computers is not as attractive in this situation as pervasive computing would be, since people would have to put on the wearable when they arrived at the library. Alternatively, the library system might be designed to "cut in" on an existing wearable. Both of these solutions seem a little intrusive. Pervasive computing, on the other hand, would allow users to interact with the catalog wherever in the library they were, rather than having to go to a place where the PC or card catalog sits. You could possibly have digital books at the end of each library shelf that gave access to the catalog.

8.3.2 Expanding the conceptual model

Considering the issues in the previous section helps the designer to envision a product. These ideas must be thought through in more detail before being prototyped

or tested with users. One aspect that will need to be decided is what technologies to use, e.g., mutimedia, virtual reality, or web-based materials, and what input and output devices best suit the situation, e.g., pen-based, touch screen, speech, keyboard, and so on. These decisions will depend on the constraints on the system, arising from the requirements you have established. For example, input and output devices will be influenced particularly by user and environmental requirements.

You also have to decide what concepts need to be communicated between the user and the product and how they are to be structured, related, and presented. This means deciding which functions the product will support, how those functions are related, and what information is required to support them. Although these decisions must be made, remember that they are made only tentatively to begin with and may change after prototyping and evaluation.

What functions will the product perform? Understanding the tasks the product will support is a fundamental aspect of developing the conceptual model, but it is also important to consider more specifically what functions the product will perform, i.e., how the task will be divided up between the human and the machine. For example, in the shared calendar example, the system may suggest dates when a set of people are able to meet, but is that as far as it should go? Should it automatically book the dates, or should it email the people concerned informing them of the meeting or asking if this is acceptable? Or is the human user or the meeting attendee responsible for checking this out? Developing scenarios, essential use cases, and use cases for the system will help clarify the answers to these questions. Deciding what the system will do and what must be left for the user is sometimes called *task* allocation. The trade-off between what to hand over to the device and what to keep in the control of the user has cognitive implications (see Chapter 3), and is linked to social aspects of collaboration (see Chapter 4). An example relating to our shared calendar system was discussed in Box 4.2 of Chapter 4: should the system allow users to book meetings in others' calendars without asking their consent first? In addition, if the cognitive load is too high for the user, then the device may be too stressful to use. On the other hand, if the device takes on too much and is too inflexible, then it may not be used at all.

Another aspect concerns the functions the hardware will perform, i.e., what functions will be hard-wired into the device and what will be left under software control, and thereby possibly indirectly in the control of the human dser? This leads to considerations of the architecture of the device, although you Would riot expect necessarily to have a clear architectural design at this stage of development.

How are the functions related to each other? Functions may be related temporally, e.g., one must be performed before another, or two can be performed in parallel. They may also be related through any number of possible categorizations, e.g., all functions relating to telephone memory storage in a cell phone, or all options for accessing files in a word processor. The relationships between tasks may constrain use or may indicate suitable task structures within the device. For example, if a task is dependent on completion of another task, then you may want to restrict the user to performing the tasks in strict order. An instance in which this has been put into

practice is in some CASE (Computer-Aided Software Engineering) tools designed to support a specific development approach. Often these tools will insist that certain diagrams must be drawn before others. For example, in object-oriented software development you normally draw class diagrams before sequence diagrams, and some tools do not allow you to draw a sequence diagram until the relevant class diagram is in place. If you're working on a small project that doesn't require this kind of discipline, this can be very frustrating, but from the perspective of a manager in charge of a large project, having these restrictions in place may be advantageous.

If task analysis has been performed on relevant tasks, the breakdown will support these kinds of decisions. For example, in the shared calendar example, the task analysis performed in Section 7.1 shows the subtasks involved and the order in which the subtasks can be performed. Thus, the system could allow meeting constraints to be found before or after the list of people, and the potential dates could be identified in the individuals' calendars before checking with the departmental calendar. It is, however, important to get both the list of attendees and meeting constraints before looking for potential dates.

What information needs to be available? What data is required to perform a task? How is this data to be transformed by the system? Data is one of the **categories** of requirements we aim to identify and capture through the requirements activity. During conceptual design, we need to consider the information requirements and ensure that our model caters for the necessary data and that information is available as required to perform the task. Detailed issues of structure and display, such as whether to use an analog display or a digital display, will more likely be dealt with in the later, physical design activity, but implications arising from the type of data to be displayed may impact conceptual design issues.

For example, in the task of booking a meeting among a set of people using the shared calendar, the system needs to be told who is to be at the meeting, how long the meeting is to take, what its location should be, and what is the latest date on which the meeting should be booked, e.g., in the next week, next two weeks, etc. In order to perform the function, the system must have this information and also must have calendar information for each of the people in the meeting, the set of locations where the meeting may take place, and ideally some way of knowing how long a person would have to travel to the location.

8.3.3 Using scenarios in conceptual design

In Chapter 7, we introduced scenarios as informal stories about user tasks and activities. They are a powerful mechanism for communicating among team members and with users. We stated in Chapter 7 that scenarios could be used and refined through different data-gathering sessions, and they can indeed be used to check out potential conceptual models.

Scenarios can be used to explicate existing work situations, but they are more commonly used for expressing proposed or imagined situations to help in conceptual design. Often, stakeholders are actively involved in producing and checking

through scenarios for a product. Bødker identifies four roles that have been suggested for scenarios (Bødker, 2000, p. 63):

- as a basis for the overall design
- for technical implementation
- as a means of cooperation within design teams
- as a means of cooperation across professional boundaries, i.e., as a basis of communication in a multidisciplinary team

In any one project, scenarios may be used for any or all of these. Box 8.4 details how different scenarios were used throughout the development of a speech-

Scenario 3: Hyper-wonderland

This scenario addresses the positive aspects of how a hypermedia solution will work.

The setting is the Lindholm construction site sometime in the future.

Kurt has access to a portable PC. The portables are hooked up to the computer at the site office via a wireless modem connection, through which the supervisors run the hypermedia application.

Action: During inspection of one of the caissons[1] Kurt takes his portable PC, switches it on and places the cursor on the required information. He clicks the mouse button and gets the master file index together with an overview of links. He chooses the links of relevance for the caisson he is inspecting.

Kurt is pleased that he no longer needs to plan his inspections in advance. This is a great help because due to the 'event-driven' nature of inspection, constructors never know where and when an inspection is taking place. Moreover, it has become much easier to keep track of personal notes, reports etc. because they can be entered directly on the spot.

The access via the construction site interface does not force him to deal with complicated keywords either. Instead, he can access the relevant information right away, literally from where he is standing.

A positive side effect concerns his reachability. As long as he has logged in on the computer, he is within reach of the secretaries and can be contacted when guests arrive or when he is needed somewhere else on the site. Moreover, he can see at a glance where his colleagues are working and get in touch with them when he needs their help or advice.

All in all, Kurt feels that the new computer application has put him more in control of things.

Scenario 4: Panopticon

This scenario addresses the negative aspects of how a hypermedia solution will work.

The setting is the Lindholm construction site sometime in the future.

Kurt has access to a portable PC. The portables are hooked up to the computer at the site office via a wireless modem connection, through which the supervisors run the hypermedia application.

Action: During inspecting one of the caissons Kurt starts talking to one of the builde n about some reinforcement problem. They argue about the recent lab tests. and he takes out his portable PC in order to provide some data which justify his arguments. It takes quite a while before he finds a spot where he can place the PC. either there is too much light, or there is no level surface at a suitable height. Finally, he puts the laptop on a big box and switches it on. He positions the cursor on the caisson he is currently inspecting and clicks the mouse to get into the master file. The table of contents pops up and from the overview of links he chooses those of relevance - but no lab test appears on the screen. Obviously, the file has not been updated as planned.

Kurt is rather upset. This loss of prestige in front of a contractor engineer would not have happened if he had planned his inspection as he had in the old days.

Sometimes, he feels like a hunted fox especially in situations where he is drifting around thinking about what kind of action to take in a particular case. If he has forgotten to log out, he suddenly has a secretary on the phone: "I see you are right at caisson 39. so could you not just drop by and take a message?"

All in all Kurt feels that the new computer application has put him under control.

[1]Used in building to hold water back during construction.

Figure 8.8 Example plus and minus scenarios.

recognition system. More specifically, scenarios have been used as scripts for user evaluation of prototypes, providing a concrete example of a task the user will perform with the product. Scenarios can also be used to build a shared understanding among team members of the kind of system being developed. Scenarios are good at selling ideas to users, managers, and potential customers. For example the scenario presented in Figure 7.7 was designed to sell ideas to potential customers on how a product might enhance their lifestyles.

An interesting idea also proposed by Bødker is the notion of *plus* and *minus scenarios*. These attempt to capture the most positive and the most negative consequences of a particular proposed design solution (see Figure 8.8) thereby helping designers to gain a more comprehensive view of the proposal.

ACTIVITY 8.5 Consider an in-car navigation device for planning routes, and suggest one plus and one minus scenario. For the plus scenario, try to think of all the possible benefits of the device. For the minus scenario, try to imagine everything that could go wrong.

Comment

Scenario 1 This plus scenario shows some potential positive aspects of an in-car navigation system.

> *"Beth is in a hurry to get to her friend's house. She jumps into the car and switches on her in-car navigation system. The display appears quickly, showing her local area and indicating the current location of her car with a bright white dot. She calls up the memory function of the device and chooses her friend's address. A number of her frequent destinations are stored like this in the device, ready for her to pick the one she wants. She chooses the "shortest route" option and the device thinks for a few seconds before showing her a bird's-eye view of her route. This feature is very useful because she can get an overall view of where she is going.*
>
> *Once the engine is started, the display reverts to a close-up view to show the details of her journey. As she pulls away from the pavement, a calm voice tells her to "drive straight on for half a mile, then turn left." After half a mile, the voice says again "turn left at the next junction." As Beth has traveled this route many times before, she doesn't need to be told when to turn left or right, so she turns off the voice output and relies only on the display, which shows sufficient detail for her to see the location of her car, her destination and the roads she needs to use."*

Scenario 2 This minus scenario shows some potential negative aspects of an in-car navigation system.

> *"Beth is in a hurry to get to her friend's house. She gets in her car and turns on the in-car navigation system. The car's battery is faulty so all the information she had entered into the device has been lost. She has to tell the device her destination by choosing from a long list of towns and roads. Eventually, she finds the right address and asks for the quickest route. The device takes ages to respond, but after a couple of minutes displays an overall view of the route it has found. To Beth's dismay, the route chosen includes one of the main roads that is being dug up over this weekend, so she cannot use the route. She needs to find another route, so she presses the cancel button and tries again to search for her friend's address through the long list of towns and roads. By this time, she is very late."*

BOX 8.4 Using Scenarios throughout Design

Scenarios were used throughout the design of a speech-recognition system (Karat, 1995). The goal of the project was to produce a product that used speech-recognition technology, so there was no defined set of user requirements to start with. The system offered speech-to-text dictation capabilities and also speech command capabilities for an application running on the same platform.

Initially, scenarios were used to set the direction of the project: discussions revolved around whether the scenario was correct or not, i.e., whether people would want to use the device to achieve the suggested task. Then scenarios were used to sketch out screens and an early user guide. Discussions at this point included checking what information was needed on the screen at what time, and also deciding what components needed to be built. Use-oriented scenarios, i.e., scenarios suggesting how the device might be used, formed the basis of early design meetings that resulted in a shared understanding of what facilities the system might include. An example scenario from basic direction setting was, "Imagine taking away the keyboard and mouse from your current workstation and describe doing everything through voice commands."

Once the basic direction was agreed, further scenarios were generated to discuss the components of the system. These scenarios focused on typical use of speech commands so that vocabulary could be tracked. An example scenario for discussing vocabulary and system components was:

> *Overall task*: Open system editor, find file REPORT.TXT, change font to Times 16, save changes, and exit the editor.

This scenario was then broken down into a specific word list as follows:

> *Voice scenario steps:* "system_editor" "open" "open" "file" "find" "r" "e" "p" "open" "font" "times" "16" "ok" "save" "close"

A short user guide was developed early on, in parallel with the initial scenario development. User guide scenarios were generated by thinking about the kinds of questions a user might need to answer, for example, What is a speech manager? How do I know what I can say?

Once early prototypes were developed, scenarios together with additional tasks were used as a basis for user testing. One of the problems was that people were unsure of what they could say, and although the system included a "What can I say?" module, this itself proved difficult to use. An example scenario used in testing was "Change the background color of the icon for the communications folder to red."

Scenarios in the form of video prototypes were taken to potential customers later in the project for feedback. The feedback they received was mostly in scenario form too, and the scenarios extracted were fed back into the design process. For example, one of the scenarios collected was, "I would like to walk around while I dictate." This could be accommodated by making mobility a factor when selecting the microphone.

Collecting feedback in the form of scenarios continued later in the project, and these informed both the design of the product and the associated documentation.

8.3.4 Using prototypes in conceptual design

The whole point of producing a prototype is to allow some evaluation of the emerging ideas to take place. As pointed out above, prototypes are built in order to answer questions. Producing anything concrete requires some consideration of the details of the design. If the prototype is to be evaluated seriously by users, then they must be able to see how their tasks might be supported by the product, and this will require consideration of more detailed aspects.

Prototyping is used to get feedback on emerging designs. This feedback may be from users, or from colleagues, or it may be feedback telling you that the idea is not technically feasible. Different kinds of prototype are therefore used at different points in the development iterations and with different people. Generally speaking, low-fidelity prototypes (such as paper-based scenarios) are used earlier in design and higher-fidelity prototypes (such as limited software implementations) are used later in design. However, low-fidelity prototypes are not very impressive to look at, so if the feedback you're looking for is approval from people who will be basing their judgment on first impressions, then a horizontal, high-fidelity prototype might suit the job better than one based on post-its or cards.

Figure 8.9 shows a card-based prototype for the shared calendar system created for a user testing session to check that the task flow and the information requirements were correct for the task of arranging a meeting. The first card shows the screen that asks the user for relevant information to find a suitable meeting date. The second card shows the screen after the system has found some potentially suitable dates and displays the results. Finally, the third screen depicts the situation

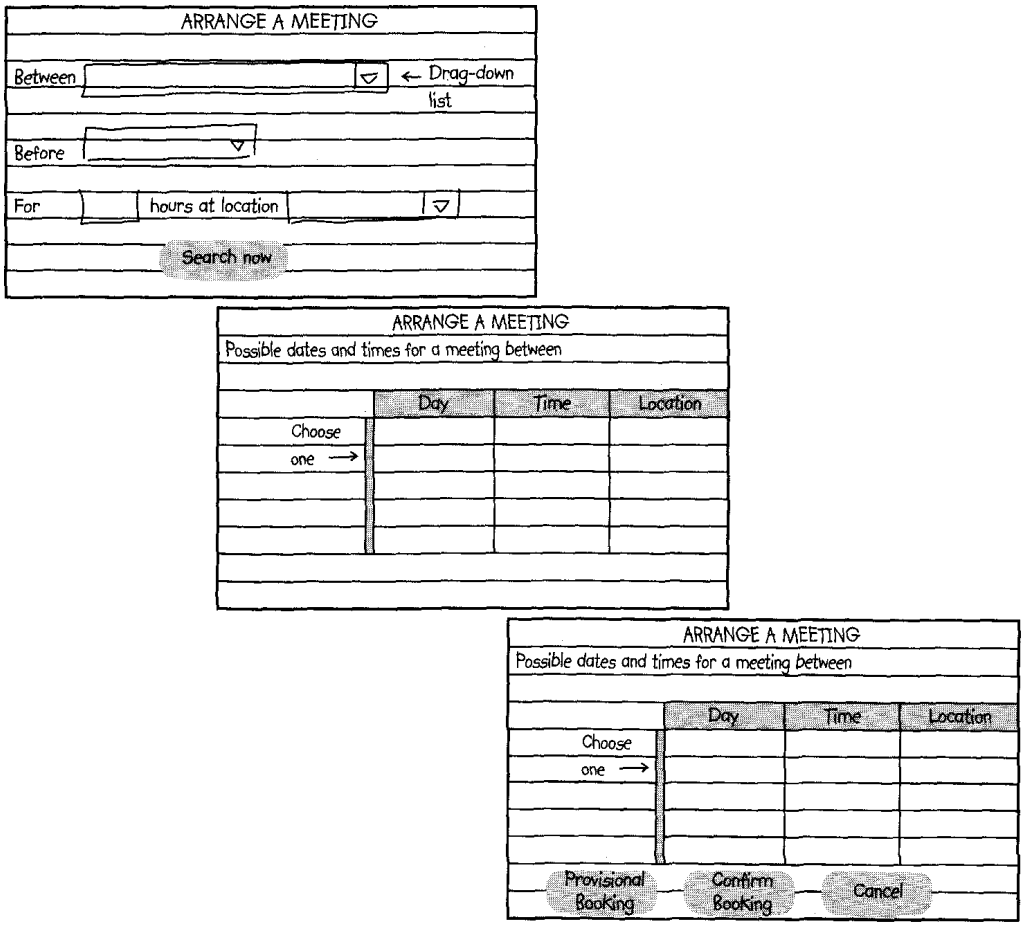


Figure 8.9 A card-based prototype for booking a meeting in the shared calendar system.

after a user has chosen one of the dates and is asked to provisionally book the chosen option, to confirm that this should be booked, or to cancel.

Note that at this point we have not decided how the navigation will work, i.e., whether there will be a tool bar, menus, etc. But we have included some detailed aspects of the design, in order to provide enough detail for users to interact with the prototype.

To illustrate how these cards can be used and the kind of information they can yield, we held a prototyping session with a potential user of the calendar. The session was informal (a kind of "quick and dirty" evaluation that you'll learn more about in Chapter 11) and lasted about 20 minutes. The user was walked through the task to see if the work flow was appropriate for the task of booking a meeting. Generally, the work flow agreed with the user's model of the task, but the session also highlighted some further considerations that did not arise in the original data gathering. Some of these had to do with work flow, but others were concerned with more detailed design. For example, the user suggested that it should be possible to state a range of dates rather than just a "before" date; he also thought that the people attending the meeting should have a chance to confirm the date through the system, and then when everyone had confirmed, the booking could be confirmed and placed in the calendar. On the detailed design, he thought that date entry through a matrix rather than a drop-down list would be more comfortable, and he asked how the possible meeting dates would be ordered. There were many more comments, all of which would be food for thought in the design. We considered only the one task, and yet it yielded a lot of very useful information.

ACTIVITY 8.6 Produce a card-based prototype for the library catalog system and the task of borrowing a book as described by the scenario, use case, and HTA in Chapter 7. You may also like to ask one of your peers to act as a user and step through the task using the prototype.

Comment Our version of the prototype is shown in Figure 8.10.

8.4 Physical design: getting concrete

Physical design involves considering more concrete, detailed issuer; of designing the interface, such as screen or keypad design, which icons to use, how to structure menus, etc.

There is no rigid border between conceptual design and physical design. As you saw above, producing a prototype inevitably means making some detailed decisions, albeit tentatively. Interaction design is inherently iterative, and so some detailed issues will come up during conceptual design; similarly.,during physical design it will be necessary to revisit decisions made during conceptual design. Exactly where the border lies is not relevant. What is relevant is that the conceptual design should be allowed to develop freely without being tied to physical constraints too early, as this might inhibit creativity.

Design is about making choices and .decisions, and the designer must strive to balance environmental, user, data and usability requirements with functional

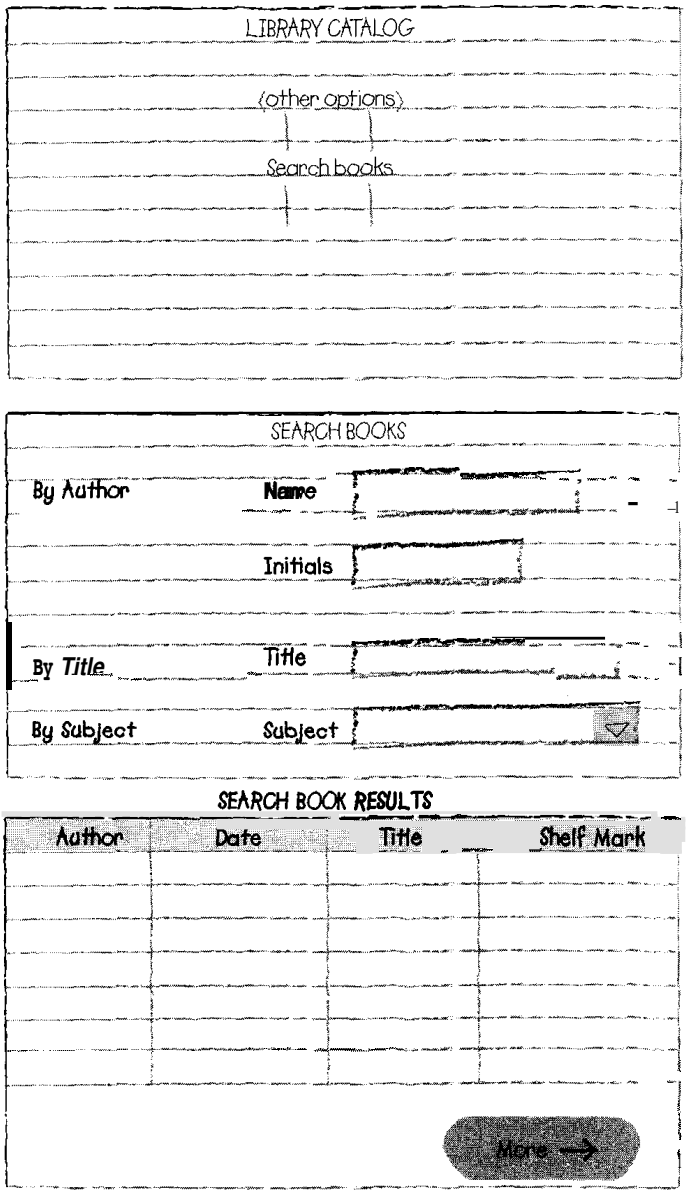


Figure 8.10 A card-based prototype for borrowing a book in the library catalog system.

requirements. These are often in conflict. For example, a cell phone must provide a lot of functionality but is constrained by having only a small screen and a small keyboard. This means that the display of information is limited and the number of unique function keys is also limited, resulting in restricted views of information and the need to associate multiple functions with function keys. Figure 8.11 shows the number of words it can display.

There are many aspects to the physical design of interactive products, and we can't cover them all in this book. Instead, we introduce some principles of

Figure 8.11 An average cell phone screen can display only a short message legibly.

good design in the context of some common interface elements. On our website (www.ID-book.com), you will find more activities and concrete examples of physical design.

8.4.1 Guidelines for physical design

The way we design the physical interface of the interactive product must not conflict with the user's cognitive processes involved in achieving the task. In Chapter 3, we introduced a number of these processes, such as attention, perception, memory, and so on, and we must design the physical form with these human characteristics very much in mind. For example, to help avoid memory overload, the interface should list options for us instead of making us remember a long list of possibilities. A wide range of guidelines, principles, and rules has been developed to help designers ensure that their products are usable, many of which are embodied in style guides and standards (see Box 8.5 for more information on this). Nielsen's set of guidelines were introduced in Chapter 1 in the form of heuristics. Another well-known set intended for informing design is Shneiderman's eight golden rules of interface design (Shneiderman, 1998):

1. *Strive for consistency.* For example, in every screen have a 'File' menu in the top left-hand corner. For every action that results in the loss of data, ask for confirmation of the action to give users a chance to change their minds.
2. *Enable frequent users to use shortcuts.* For example, in most word-processing packages, users may move around the functions using menus or shortcut "quick keys," or function buttons.
3. *Offer informative feedback.* Instead of simply saying "Error 404," make it clear what the error means: "The URL is unknown." This feedback is also influenced by the kinds of users, since what is meaningful to a scientist may not be meaningful to a manager or an architect.
4. *Design dialogs to yield closure.* For example, make it clear when an action has completed successfully: "printing completed."
5. *Offer errorprevention and simple error handling. It* is better for the user not to make any errors, i.e., for the interface to prevent users from making mistakes. However, mistakes are inevitable and the system should be forgiving about the errors made and support the user in getting back on track.
6. *Permit easy reversal of actions.* For example, provide an "undo" key where possible.
7. *Support internal locus of control.* Users feel more comfortable if they feel in control of the interaction rather than the device being in control.

8. ***Reduce short-term memory load.*** For example, wherever possible, offer users options rather than ask them to remember information from one screen to another.

Other guidelines that have been suggested include keeping the interaction simple and clear, organizing interface elements to aid understanding and use through suitable groupings, and designing images to be immediate and generalizable. All of

BOX 8.5 Design Guidelines and Standards

Design guidelines and standards exist to help designers create better designs by learning from others' experience. Some guidelines are at a detailed level and are called design rules, while others are more abstract, require interpretation before being applied, and are called design principles. For example, one very general but very pertinent guideline for website design is, "Keep it simple." This is relevant throughout design but must be interpreted within the specific context before it can be applied. On the other hand, a design rule for web design might be, "Don't offer an option to search the whole web from your own website." This is a very specific rule that requires no interpretation to apply. These terms were introduced in Box 1.5 of Chapter 1, together with some others commonly used in this context.

Design Principles

Principles often embody design-related information derived from theory, and so this is one way in which the cognitive models and processes introduced in Chapter 3 can be put to practical use in your designs. For example, the guideline to use "recognition rather than recall" is based on theories of memory that say people find it is easier to recognize things than to remember them with no prompting. However carefully names are chosen to reflect function, it is easier to choose the right option from a list of options than to remember the name of a command. Shneiderman's (1998) eight golden rules of interface design are an example set of principles.

Rules

Rules are more specific versions of design guidelines and provide more detailed guidance. A classic example of design rules is the collection by Smith and Mosier (1986). These rules are quite detailed and prescriptive. For example, one dealing with consistent format states, "Adopt a consistent format for the location of various display features from one display to another." Each rule is accompanied by explanatory notes such as examples, exceptions, comments that provide detailed guidance, and any useful references to their own or others' work.

Style guides

A style guide is a collection of specific design rules and principles from which the rules are derived. They are used to ensure a consistent look and feel across a set of applications. The most widely known style guides are those for Windows development (Microsoft Corporation, 1992) and for Macintosh development (Apple Computer Inc., 1993). An example from the Macintosh human interface guidelines concerned with designing color icons states that "When you design an icon, you should start by creating the black-and-white version first, then the color should be added."

Style guides tailored to a specific company can be used to deliver a particular corporate image. Such style guides are called corporate style guides.

Standards

Some international standards govern the development of interactive systems. These are also collections of principles and rules to provide designers with a framework based on others' experience. The most pertinent standards are:

- ISO9241 Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs)
- ISO 13407 Human-centered Design Processes for Interactive Systems.
- ISO 14915 Design of the User Interface of Multimedia Applications.

these focus on making the communication between user and product as clear as possible.

Extensive experience in the art of communication (through posters, text, books, images, advertising, etc.) is relevant to interaction design. In her interview at the end of Chapter 6, Gillian Crampton Smith identifies the roles that traditional designers can play in interaction design; one of them she highlights is the fact that designers are trained to produce a coherent design that delivers the desired message to the intended audience. Including such designers on the team can bring this experience to bear. Mullet and Sano (1995) identify a number of useful design principles arising from the visual arts.

To see how these can be translated into the context of interaction design, we consider their application to different widgets, i.e., screen elements, in the next section.

8.4.2 Different kinds of widget

Interfaces are made up of widgets, elements such as dialog boxes, menus, icons, toolbars, etc. Each element must be designed or chosen from a predesigned set of widgets. Sometimes these decisions are made for you through the use of a style guide. Style guides may be commercially produced, such as the Windows style guide (called commercial style guides), or they may be internal to a company (called corporate style guides). A style guide dictates the look and feel of the interface, i.e., which widgets should be used for which purpose and what they look like. For example, study your favorite Windows applications. Which menu is always on the right-hand side of the toolbar? What icon is used to represent "close" or "print"? Which typeface is used in menus and dialog boxes? Each Windows product has the same look and feel, and this is specified in the Windows style guide. If you go to a commercial website, you may find that each screen also has the same look and feel to it. This kind of corporate identity can be captured in a corporate style guide. More information about standards and style guides is in Box 8.5.

We consider here briefly three main aspects of interface design: menu design, icon design, and screen layout. These are applicable to a wide range of interactive products, from standard desktop interfaces for PC software, to mobile communicator functions and microwave ovens.

Menu design Menus provide users with a choice that can be a choice of commands or a choice of options related to a command. They provide the means by which the user can perform actions related to the task in hand and therefore are based on task structure and the information required to perform a task.

Menus may be designed as drop-down, pop-up or single-dialog menus. It may seem obvious how to design a menu, but if you want to make the application easy to use and provide user satisfaction, some important points must be taken into account. For example, for pull-down and pop-up menus, the most commonly used functions should be at the top, to avoid frequent long scans and scrolls. The principle of grouping can be used to good effect in menu design. For example, the menu can be divided into collections of items that are related, with each collection being

5.2 Grouping options in a menu

Menu options should be grouped within a menu to reflect user expectations and facilitate option search.

5.2.1 Logical groups

If the menu option contains a large number of options (eight or more) and these options can be logically grouped, options should be grouped by function or into other logical categories which are meaningful to users.

EXAMPLE: Grouping the commands in a word-processing system into such categories as customise, compose, edit, print.

5.2.2 Arbitrary groups

If 8 or more options are arranged arbitrarily in a menu panel, they should be arranged into equally distributed groups utilising the following equation:

$$g = \sqrt{n}$$

where

g is the number of groups
n is the number of options on the panel.

EXAMPLE: Given 19 options in a menu panel, arrange them into 4 groups of about 5 options each.

Figure 8.12 An excerpt from ISO 9241 concerning how to group items in a menu.

separated from others. Opposite operations such as "quit" and "save" should be clearly separated to avoid accidentally losing work instead of saving it (See Figure 1.6 in Chapter 1).

An excerpt from ISO 9241, a major international standard for interaction design, considers grouping in menu design, as shown in Figure 8.12.

To show how the design of menus may proceed, we return to the shared calendar. In our initial data gathering, we identified a number of possible tasks that the user might want to perform using the calendar. These included making an entry, arranging a meeting among a number of people, entering contact details, and finding out other people's engagements. Tied to these would also be a number of administrative and housekeeping actions such as deleting entries, moving entries, editing entries, and so on. Suppose we stick with just this list. The first question is what to call the menu entries. Menu names need to be short, clear, and unambiguous. The space for listing them will be restricted, so they must be short, and you want them to be distinguishable, i.e., not easily confused with one another so that the user won't choose the wrong one by mistake. Our current descriptions are really too long. For example, instead of "find out other people's engagements" we could have ***Query entry*** as a menu option, following through to a dialog box that asks for relevant details.

We need to consider logical groupings. In this case, we could group according to user goal, i.e., have ***Query entry, Add entry, Edit entry, Move entry,*** and ***Delete entry*** grouped together (see Figure 8.13). Similarly, we could group Add ***contact,***

Calendar Entry	Contacts	Arrange Meeting
Add Entry	Add Contact	
Edit Entry	Edit Contact	
Move Entry	Delete Contact	
Delete Entry		

Figure 8.13 Possible menu groupings for the shared calendar system.

Edit contact and ***Delete contact*** together. Finding other people's engagements could be generalized to a simple ***Search*** option that led to a dialog box in which the search parameters are specified. Arranging a meeting is also an option that doesn't clearly group with other commands. This and the ***Search*** option may be better represented as options on a toolbar than as menu items on their own.

Icon design Designing a good icon takes more than a few minutes. You may be able to think up good icons in a matter of seconds, but such examples are unlikely to be widely acceptable to your user group. When symbols for representing ladies' and gents' toilets first appeared in the UK, a number of confused tourists did not understand the culturally specific icons of a woman wearing a skirt and a man wearing trousers. For example, some people protested that they thought the male icon was a woman wearing a trouser suit. We are now all used to these symbols, and indeed internationally recognized symbols for how to wash clothes, fire exits, road signs, etc. now exist. However, icons are cultural and context-specific. Designing a good icon takes time.

At a simple level, designers should always draw on existing traditions or standards, and certainly should not contradict them. Concrete objects or things are easier to represent as an icon since they can be just a picture of the item. Actions are harder but can sometimes be captured. For example, using a picture of a pair of scissors to represent "cut" in a word-processing application provides sufficient clues as long as the user understands the convention of "cut" for deleting text.

In our shared calendar, if we are going to have the ***Search*** and ***Arrange a Meeting*** commands on a tool bar, we need to identify a suitable icon for each of them. A number of possible icons spring to mind for the ***Search*** option, mainly because searching is a fairly common action in many interactive products: a magnifying glass or a pair of binoculars are commonly used for such options. Arranging a meeting is a little difficult, though. It's probably easier to focus on the meeting itself than the act of arranging the meeting, but how do you capture a meeting? You want the icon to be immediately recognizable, yet it must be small and simple. What characteristic(s) of a meeting might you capture? One of the things that comes to mind is a group of people, so maybe we could consider a collection of stick people? Another element of a meeting is usually a table, but a table on its own isn't enough, so maybe having a table with a number of people around it would work?

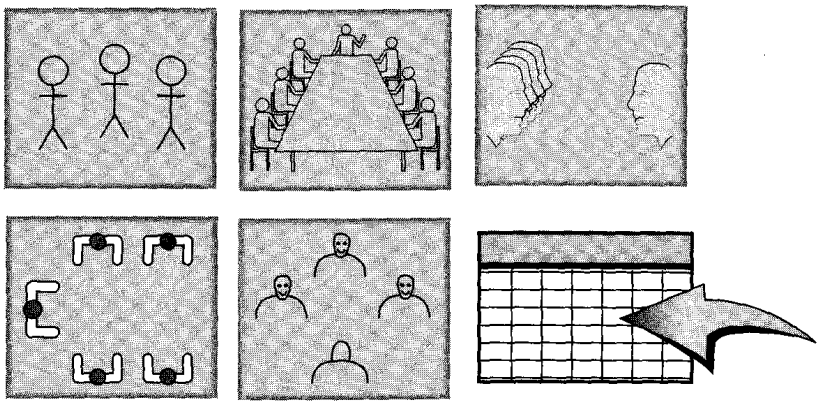


Figure 8.14 **A** variety of possible icons to represent the "arrange a meeting" function.

ACTIVITY 8.7 Sketch a simple, small icon to represent a set of people around the table, or suggest an icon of your own. Show it to your peers or friends, tell them that it's an icon for a shared calendar application, and see if they can understand what it represents.

Comment **A** variety of attempts are shown in Figure 8.14. The last icon is the icon that paim.net uses for arranging meetings. This is a different possibility that tries to capture the fact that you're entering data into the planner.

We discussed some cognitive aspects relevant to icon design in Chapter **3.** For example, icons must be designed so that users can readily perceive their meaning and so that they are distinguishable one from another. Since the size of icons on the screen is often very small, this can be difficult to achieve, but users must be able to tell them apart. Look back again at Figure 3.4 and the activity associated with it. How easy do you think it would be to tell some of these icons apart if they were just a little smaller, or the screen resolution was lower?

Screen *design.* There are two aspects to screen design: how the task is split across a number of screens, and how the individual screens are designed.

The first aspect can be supported by reference to the task analysis, which broke down the user's task into subtasks and plans of action. One starting point for screen design is to translate the task analysis into screens, so that each task or subtask has its own screen. This will require redesign and adjustment, but it is a starting point. The interaction could be divided into simple steps, each involving a decision or simple data entry. However, this can become idiotic, and having too many simple screens can become just as frustrating as having information all crammed into one screen. This is one of the balances to be drawn in screen design. Tasks that are more complicated than this (and are usually unsuited to simple task analysis) may require a different model of interaction in which a number of screens are open at the same time and the user is allowed to switch among them.

Another issue affecting the division of a task across screens is that all pertinent information must be easily available at relevant times.

Guidelines for the second aspect, individual screen design, draw more clearly from some of the visual communication principles we mentioned above: for example, designing the screen so that users' attention is drawn immediately to the salient points, and using color, motion, boxing and grouping to aid understanding and clarity. Each screen should be designed so that when users first see it, their attention is focused on something that is appropriate and useful to the task at hand. Animations can be very distracting if they are not relevant to the task, but are effective if used judiciously.

Good organization helps users to make sense of an interaction and to interpret it within their own context (as discussed in Chapter **3).** This is another example where principles of good grouping can be applied, for example, grouping similar things together or providing separation between dissimilar or unrelated items. Grouping can be achieved in different ways: by placing things close together, using colors, boxes, or frames to segregate items, or using shapes to indicate relationships among elements. There is a trade-off between sparsely populated screens with a lot of open space and overcrowded screens with too many and too complicated sets of icons. If the screen is overcrowded, then users

BOX 8.6 Design Patterns for HCI

Design patterns have become popular in software engineering since the early 1990s. Patterns capture experience, but they have a different structure and a different philosophy from other forms of guidance, such as the guidelines we introduced earlier, or specific methods. One of the intentions of the patterns community is to create a vocabulary, based on the names of the patterns, that designers can use to communicate with one another and with users. Another is to produce a literature in the field that documents experience in a compelling form.

The idea of patterns was first proposed by Christopher Alexander, a British architect who described patterns in architecture. His hope was to capture the "quality without a name" that is recognizable in something when you know it's good.

But what is a pattern? One simple definition is that it is a solution to a problem in a context. What this means is that a pattern describes a problem, a solution, and where this solution has been found to work. This means that users of the pattern can see not only the problem and solution, but also understand when and where it has worked before and access a rationale for why it worked. This helps designers in adopting it (or not) for themselves.

Patterns on their own are interesting, but are not as powerful as a pattern language. A pattern language is a network of patterns that reference one another and work together to create a complete structure.

The application of patterns in HCI is still in its infancy. But some work has been done in the area, and some pattern languages have been proposed. One of the most mature languages is that described by Jan Borchers (2001) for interactive music exhibits. Borchers presents three related languages: one for music, one for HCI and one for software engineering, all of which have arisen from his experience of designing music exhibits. The HCI language addresses issues such as accommodating groups as well as single users, handling complexity, content structure, and interaction devices.

BOX 8.7 Designing for the Web

Web pages need to exhibit the kinds of good interaction design we talk about in this chapter, but they also have some specific requirements. For example, Nielsen (2000) has suggested a set of evaluation criteria specifically for the web (see Chapter 13 for more detail).

The key design issues for websites that are different from other interaction designs are captured very well by three questions proposed by Keith Instone (quoted in Veen, 2001): Where am I? What's here? Where can I go? Each web page should be designed with these three questions in mind. The answers must be clear to users. Jeffrey Veen (2001) expands on these questions. He suggests that a simple way to view a web page is to deconstruct it into three areas (see Figure 8.15). Across the top would be the answer to "Where am I?" Because users can arrive at a site from any direction (and rarely through the front door, or home page), telling them where they are is critical. Having an area at the top of the page that "brands" the page instantly provides that information. Down the left-hand side is an area in which navigation or menus sit. This should allow users immediately to see what else is available on the site, and answers the question "Where can I go?'

The most important information, and the reason a user has come to the site in the first place, is provided in the third area, the content area, which answers the question "What's here?" Content for web pages must be designed differently from standard documents, since the way users read web pages is different. On web pages, content should be short and precise, with crisp sentences. Using headlines to capture the main points of a paragraph is one way to increase the chances of your message getting over to a user who will, perhaps, merely scan the page rather than look at it in detail.

Branding a page is important for the reasons given above. We have also talked about the need to keep screens uncluttered so that people can find their way around and see clearly what is available. However, there may be occasions when the need to maintain a brand overrides other design issues. For example, the website for the Swedish newspaper *Aftonbladet*, while quite busy and crowded (see Figure 8.16), was designed to continue the style of the paper-based version, which also has a busy and crowded appearance.

Download times are critical for the success of websites. Users who have to wait too long for a page will move on somewhere else. So although it's attractive to have graphics on your pages, use them sparingly. One suggestion from Nielsen (2000) is to have few graphics on the welcoming or more abstract pages, and offer users the chance to see pictures of products, or maps, or whatever, only when they explicitly ask for them. It is quite common to use thumbnails, miniaturized versions of the full picture, as links.

Traditionally, hyperlinks have been indicated by highlighting the text in blue and underlining it.

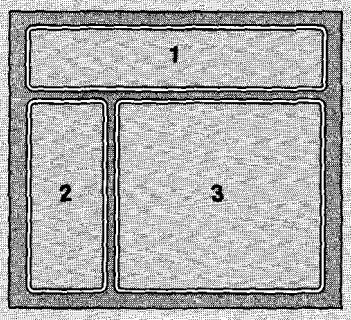


Figure 8.15 Any web page has three main areas.

Figure 8.16 The front web page of the Aftonbladet newspaper.

This is how links were indicated in hypertext, and this convention transferred to web pages. However, now that web pages can be designed more flexibly, some sites are moving away from the traditional blue underlining and are using "roll-over" tags to signify links, i.e., when the mouse rolls over a word then a label appears. This lack of consistency, together with the fact that underlined text is not always a link, can cause considerable confusion for the user.

will become confused and distracted. But too much open space and consequently many screens can lead to frequent screen changes, and a disjointed series of interactions.

information *display*. Making sure that the relevant information is available for the task is one aspect of information display, but another concerns the format. Differ-

ent types of information lend themselves to different kinds of display. For example, data that is discrete in nature, such as sales figures for the last month, could be displayed graphically using a digital technique, while data that is continuous in nature, such as the percentage increase in sales over the last month, is better displayed using an analog device.

If data is to be transferred to the device from a paper-based medium or vice versa, it makes sense to have the two consistent. This reduces user confusion and search time in reconciling data displayed with data on the paper.

In the shared calendar application, there is potentially a lot of information to display. If you have five members of the department, each with their own calendars, and the departmental calendar too, then you need to display six sets of engagement information. When we showed the prototype system to our user, he suggested that dates should be chosen through a matrix of some kind rather than a drop-down list. Displaying information appropriately can make communication a lot easier.

8.5 Tool support

The tools available to support the activities described here are wide-ranging and various. We mentioned development environments when talking about prototypes in Section 8.2, but other kinds of support are available.

Much research has been done into appropriate support for different kinds of design and software production, resulting in a huge variety of tools. Because technology moves so quickly, any discussion of specific tools would be quickly out of date. Up-to-date information about support tools can be found on our website (www.ID-book.com). Here we report on some general observations about software tools.

Brad Myers (1995) suggests nine facilities that user interface software tools might provide:

- help design the interface given a specification of the end users' tasks
- help implement the interface given a specification of the design
- create easy-to-use interfaces
- allow the designer to rapidly investigate different designs
- allow nonprogrammers to design and implement user interfaces
- automatically evaluate the interface and propose improvements
- allow the end user to customize the interface
- provide portability
- be easy to use

In a later paper Myers et al. (2000), look at the past, present, and future of user interface tools. Box 8.8 describes some types of tool that have been successful and some that have been unsuccessful.

BOX 8.8 Successes and Failures for User Interface Tools *(Myers et al., 2000)*

Looking at the history of user interface design tools, we can see some tools that have been successful and have withstood the test of time, and others that have fallen by the wayside. Understanding something of what works and what doesn't gives us lessons for the future of such tools.

Tools that have been successful are:

Window Managers and Toolkits. The idea of overlapping windows was first proposed by Alan Kay (Kay, 1969). These have been successful because they help to manage scarce resources: screen space and human perceptual and cognitive resources such as limited visual field and attention.

Event languages that are designed to program actions based on external events: for example, when the left mouse button is depressed, move the cursor here. These have worked because they map well to the direct manipulation graphical user interface style.

Interactive graphical tools or interface builders, such as Visual Basic. These allow the easy construction of user interfaces by placing interface elements on a screen using a mouse. They have been successful because they use a graphical means to design a graphical layout, i.e., you can build a graphical screen layout by grabbing and placing graphical elements without touching any program code.

Component systems are based on the idea of dynamically combining individual components that have been separately written and compiled. Sun's Java Beans uses this approach. One reason for its success is that it addresses the important software engineering goal of modularity.

Scripting languages have become popular because they support fast prototyping. Example scripting languages are Python and Perl.

Hypertext allows elements of a document to be linked in a multitude of ways, rather than the traditional linear layout. Most people are aware of hypertext links because of their use on the web.

Object-oriented programming. This programming approach is successful in interface development because the objects of an interface such as buttons and other widgets can so readily be cast as objects in the language.

Promising approaches that have not caught on are:

Technology has changed so fast that in some cases the tools to support the development of certain technologies have failed to keep up with the rapidly changing requirements. Good ideas that have fallen by the wayside include:

User interface management tools (UIMS). The idea behind UIMS was akin to the idea behind database management systems. Their purpose was to abstract away the details of interface implementation to allow developers to specify and manipulate interfaces at a higher level of abstraction. This separation turned out to be undesirable, as it is not always appropriate to be able to understand and manipulate interface elements only at a high level of abstraction.

Formal language based tools. Many systems in the 1980s were based on formal language concepts such as state transition diagrams and parsers for context-free grammars. These failed to catch on because: the dialog-based interfaces for which these tools were particularly suited were overtaken by direct manipulation interfaces; they were very good at producing sequential interfaces, but not at expressing unordered sequences of action; and they were difficult to learn even for programmers.

Constraints. Tools that were designed to maintain constraints, i.e., relationships among elements of an interface such as that the scroll bar should always be on the right of the window, or that the color of one item should be the same as the color of other items. These systems have not caught on because they can be unpredictable. Once constraints are set up, the tool must find a solution to maintain them, and since there is more than one solution, the tool may find a solution the user didn't expect.

Model-based and automatic techniques. The aim of these systems was to let developers specify interfaces at a high level of abstraction and then for an interface to be automatically generated according to a predefined set of interpretation rules. These too have suffered from problems of unpredictability, since the generation of the interfaces relies on heuristics and rules that are themselves unpredictable in concert.

Assignment

This Assignment continues work on the web-based ticket reservation system at the end of Chapter 7.

(a) Based on the information gleaned from the assignment in Chapter 7, suggest three different conceptual models for this system. YOU should consider each of the aspects of a conceptual model discussed in this chapter: interaction paradigm, interaction mode, metaphors, activities it will support, functions, relationships between functions, and information requirements. Of these, decide which one seems most appropriate and articulate the reasons why.

(b) Produce the following prototypes for your chosen conceptual model.

(i) Using the scenarios generated for the ticket reservation system, produce a storyboard for the task of buying a ticket for one of your conceptual models. Show it to two or three potential users and get some informal feedback.

(ii) Now develop a prototype based on cards and post-it notes to represent the structure of the ticket reservation task, incorporating the feedback from the first evaluation. Show this new prototype to a different set of potential users and get some more informal feedback.

(iii) Using a software-based prototyping tool (e.g., Visual Basic or Director) or web authoring tool (e.g., Dreamweaver), develop a software-based prototype that incorporates all the feedback you've had so far. If you do not have experience in using any of these, create a few HTML web pages to represent the basic structure of your website.

(c) Consider the web page's detailed design. Sketch out the application's main screen (home page or data entry). Consider the screen layout, use of colors, navigation audio, animation, etc. While doing this, use the three main questions introduced in Box 8.7 as guidance: Where am I? What's here? Where can I go? Write one or two sentences explaining your choices, and consider whether the choice is a usability consideration or a user experience consideration.

Summary

This chapter has explored the activities of design prototyping and construction. Prototyping and scenarios are used throughout the design process to test out ideas for feasibility and user acceptance. We have looked at the different forms of prototyping, and the activities have encouraged you to think about and apply prototyping techniques in the design process.

Key points

- Prototyping may be low fidelity (such as paper-based) or high fidelity (such as software-based).
- High-fidelity prototypes may be vertical or horizontal.
- Low-fidelity prototypes are quick and easy to produce and modify and are used in the early stages of design.
- There are two aspects to the design activity: conceptual design and physical design.
- Conceptual design develops a model of what the product will do and how it will behave, while physical design specifies the details of the design such as screen layout and menu structure.

- **We have explored three perspectives to help you develop conceptual models: an interaction paradigm point of view, an interaction mode point of view, and a metaphor point of view.**
- **Scenarios and prototypes can be used effectively in conceptual design to explore ideas.**
- **We have discussed four areas of physical design: menu design, icon design, screen design, and information display.**
- **There is a wide variety of support tools available to interaction designers.**

Further reading

WINOGRAD, TERRY (1996) ***Bringing Design to Software.*** Addison-Wesley and ACM Press. This book is a collection of articles all based on the theme of applying ideas from other design disciplines in software design. It has a good mixture of interviews, articles, and profiles of exemplary systems, projects or techniques. Anyone interested in software design will find it inspiring.

CARROLL, JOHN M. (ed.) (1995) ***Scenario-based Design.*** John Wiley & Sons, Inc. This volume is an edited collection of papers arising from a three-day workshop on use-oriented design. The book contains a variety of papers including case studies of scenario use within design, and techniques for using them with object-oriented development, task models and usability engineering. This is a good place to get a broad understanding of this form of development.

MULLET, KEVIN, AND SANO, DARELL (1995) ***Designing Visual Interfaces.*** SunSoft Press. This book is full of practical guidance for designing interactions that focus on communication. The ideas here come from communication-oriented visual designers. Mullet and Sano show how to apply these techniques to interaction design, and they also show some common errors made by interaction designers that contravene the principles.

VEEN, JEFFREY (2001) ***The Art and Science of Web Design.*** New Riders. A very bright book, providing a lot of practical information taken from the visual arts about how to design websites. It also includes sections on common mistakes to help you avoid these pitfalls.

MYERS, BRAD, HUDSON, S. E., AND PAUSCH, R. (2000) Past, present and future of user interface software tools. ACM ***Transactions on Computer-Human Interaction,*** 7(1), ***3–28.*** This paper presents an interesting description of user interface tools, expanding on the information given in Box 8.8.