•52•

PROTOTYPING TOOLS AND TECHNIQUES

Michel Beaudouin-Lafon
Université Paris-Sud

Wendy E. Mackay
INRIA

Introduction **1018**
What Is a Prototype? 1018
Prototypes as Design Artifacts **1018**
Representation 1018
Precision 1019
Interactivity 1019
Evolution 1020
Prototypes and the Design Process **1020**
User-Centered Design 1020
Participatory Design 1021
Exploring the Design Space 1021
Expanding the Design Space: Generating Ideas 1022
Contracting the Design Space:
Selecting Alternatives 1023
Prototyping Strategies 1024
Horizontal prototypes 1024
Vertical prototypes 1024
Task-oriented prototypes 1024
Scenario-based prototypes 1024
Rapid Prototyping **1025**
Offline Rapid Prototyping Techniques 1025
Paper and pencil 1025
Mock ups 1025
Wizard of Oz 1026
Video prototyping 1026
Online Rapid Prototyping Techniques 1028
Noninteractive simulations 1029
Interactive simulations 1030
Scripting languages 1030
Iterative and Evolutionary Prototypes **1032**
User Interface Toolkits 1033
User Interface Builders 1033
User Interface Development Environments 1034
Prototyping Mixed Reality and
Pervasive Computing Systems **1036**
Conclusion **1037**
References **1038**

INTRODUCTION

"A good design is better than you think."
(Heftman, as cited in Raskin, 2000).

Design is about making choices. In many fields that require creativity and engineering skill, such as architecture and automobile design, prototypes both inform the design process and help designers select the best solution.

This chapter describes tools and techniques for using prototypes to design interactive systems. The goal is to illustrate how they can help designers generate and share new ideas, get feedback from users or customers, choose among design alternatives and articulate reasons for their final choices.

We begin with our definition of a prototype and then discuss prototypes as design artifacts, introducing four dimensions for analyzing them. We then discuss the role of prototyping within the design process, in particular the concept of a design space and of how it is expanded and contracted by generating and selecting design ideas. The following sections describe specific prototyping approaches: rapid prototyping, both offline and online, for early stages of design, and iterative and evolutionary prototyping, which use online development tools. Finally, we address the specific issue of prototyping mixed reality and pervasive computing systems.

What Is a Prototype?

We define a *prototype* as a concrete representation of part or all of an interactive system. A prototype is a tangible artifact, not an abstract description that requires interpretation. Designers, as well as managers, developers, customers and end users, can use these artifacts to envision and to reflect upon the final system.

Note that other fields may define prototype differently. For example, an architectural prototype is a scaled-down model of the final building. This is not possible for prototypes of interactive systems: The designer may limit the amount of information the prototype can handle, but the actual user interface must be presented at full scale. Thus, a prototype interface to a database may handle only a small subset of the final database but must still present a full-size display and interaction techniques. Fashion designers create another type of prototype, a full-scale, one-of-a-kind model, such as a handmade dress sample. Although in haute couture, this prototype may also be the final product, the ready-to-wear market requires additional design phases to create a design that can be mass-produced in a range of sizes. Some interactive system prototypes begin as one-of-a-kind models that are then distributed widely (since the cost of duplicating software is so low). However, most successful software prototypes evolve into the final product and then continue to evolve as new versions of the software are released.

Hardware and software engineers often create prototypes to study the feasibility of a technical process. They conduct systematic, scientific evaluations with respect to predefined benchmarks and, by systematically varying parameters, fine tune the system. Designers in creative fields, such as typography or graphic design, create prototypes to express ideas and reflect on them. This approach is intuitive, oriented more to discovery and generation of new ideas than to evaluation of existing ideas.

HCI is a multidisciplinary field that combines elements of science, engineering, and design (Mackay & Fayard, 1997; Dijkstra-Erikson et al., 2001). Prototyping is primarily a design activity, although we use software engineering to ensure that software prototypes evolve into technically sound working systems and we use scientific methods to study the effectiveness of particular designs.

PROTOTYPES AS DESIGN ARTIFACTS

We can look at prototypes as both concrete artifacts in their own right or as important components of the design process. When viewed as artifacts, successful prototypes have several characteristics: They support creativity, helping the developer to capture and generate ideas, facilitate the exploration of a design space, and uncover relevant information about users and their work practices. They encourage communication, helping designers, engineers, managers, software developers, customers and users to discuss options and interact with each other. They also permit early evaluation since they can be tested in various ways, including traditional usability studies and informal user feedback, throughout the design process.

We can analyze prototypes and prototyping techniques along four dimensions:

- *Representation* describes the form of the prototype, such as sets of paper sketches or computer simulations;
- *Precision* describes the level of detail at which the prototype is to be evaluated, such as informal and rough or highly polished;
- *Interactivity* describes the extent to which the user can actually interact with the prototype, such as "watch only" or fully interactive;
- *Evolution* describes the expected life cycle of the prototype, such as throwaway or iterative.

Representation

Prototypes serve different purposes and thus take different forms. A series of quick sketches on paper can be considered a prototype; so can a detailed computer simulation. Both are useful; both help the designer in different ways. We distinguish between two basic forms of representation: offline and online.

Offline prototypes (also called "paper prototypes") do not require a computer. They include paper sketches, illustrated storyboards, cardboard mock ups and videos. The most salient characteristics of offline prototypes (of interactive systems) is that they are created quickly, usually in the early stages of design, and they are usually thrown away when they have served their purposes.

Online prototypes (also called "software prototypes") run on a computer. They include computer animations, interactive video presentations, programs written with scripting languages, and applications developed with interface builders. The cost of

producing online prototypes is usually higher, and may require skilled programmers to implement advanced interaction and visualization techniques or to meet tight performance constraints. Software prototypes are usually more effective in the later stages of design, when the basic design strategy has been decided.

In our experience, programmers often argue in favor of software prototypes even at the earliest stages of design. Because they are already familiar with a programming language, these programmers believe it will be faster and more useful to write code than to "waste time" creating paper prototypes. In 20 years of prototyping, in both research and industrial settings, we have yet to find a situation in which this is true.

First, offline prototypes are very inexpensive and quick. These permit a very rapid iteration cycle and help prevent the designer from becoming overly attached to the first possible solution. Offline prototypes make it easier to explore the design space, examining a variety of design alternatives and choosing the most effective solution. Online prototypes introduce an intermediary between the idea and the implementation, slowing down the design cycle.

Second, offline prototypes are less likely to constrain how the designer thinks. Every programming language or development environment imposes constraints on the interface, limiting creativity and restricting the number of ideas considered. If a particular tool makes it easy to create scroll-bars and pull-down menus and difficult to create a zoomable interface, the designer is likely to limit the interface accordingly. Considering a wider range of alternatives, even if the developer ends up using a standard set of interface widgets, usually results in a more effective design.

Finally, and perhaps most importantly, offline prototypes can be created by a wide range of people, not just programmers. Thus all types of designers, technical or otherwise, as well as users, managers, and other interested parties, can all contribute on an equal basis. Unlike programming software, modifying a storyboard or cardboard mock up requires no particular technical skill. Collaborating on paper prototypes not only increases participation in the design process, but also improves communication among team members and increases the likelihood that the final design solution will be well accepted.

Although we believe strongly in offline prototypes, they are not a panacea. In some situations, they are insufficient to evaluate fully a particular design idea. For example, interfaces requiring rapid feedback to users or complex, dynamic visualizations usually require software prototypes. However, particularly when using video and Wizard of Oz techniques, offline prototypes can be used to create very sophisticated representations of the system.

Prototyping is an iterative process and all prototypes provide information about some aspects while ignoring others. The designer must consider the purpose of the prototype (Houde & Hill, 1997) at each stage of the design process and choose the representation that is best suited to the current design question.

Precision

Prototypes are explicit representations that help designers, engineers and users reason about the system being built. By their nature, prototypes require details. A verbal description such as "the user opens the file" or "the system displays the results" provides no information about what the user actually does or sees. Prototypes force designers to show the interaction: just how does the user open the file and what are the specific results that appear on the screen?

Precision refers to the relevance of details with respect to the purpose of the prototype.[1] For example, when sketching a dialog box, the designer specifies its size, the positions of each field and the titles of each label. However not all these details are relevant to the goal of the prototype: it may be necessary to show where the labels are, but too early to choose the text. The designer can convey this by writing nonsense words or drawing squiggles, which shows the need for labels without specifying their actual content.

Although it may seem contradictory, a detailed representation need not be precise. This is an important characteristic of prototypes: those parts of the prototype that are not precise are those open for future discussion or for exploration of the design space. Yet they need to be incarnated in some form so the prototype can be evaluated and iterated.

The level of precision usually increases as successive prototypes are developed and more and more details are set. The forms of the prototypes reflect their level of precision: sketches tend not to be precise, whereas computer simulations are usually very precise. Graphic designers often prefer using hand sketches for early prototypes because the drawing style can directly reflect what is precise and what is not: the wiggly shape of an object or a squiggle that represents a label are directly perceived as imprecise. This is more difficult to achieve with an online drawing tool or a user interface builder.

The form of the prototype must be adapted to the desired level of precision. Precision defines the tension between what the prototype states (relevant details) and what the prototype leaves open (irrelevant details). What the prototype states is subject to evaluation; what the prototype leaves open is subject to more discussion and design space exploration.

Interactivity

An important characteristic of HCI systems is that they are interactive: users both respond to them and act upon them. Unfortunately, designing effective interaction is difficult: many interactive systems (including many websites) have a good look but a poor feel. HCI designers can draw from a long tradition in visual design for the former, but have relatively little experience with how interactive software systems should be used: personal computers have only been commonplace for a couple decades. Another problem is that the quality of interaction is tightly linked to the end users and a deep understanding of their work prac-

[1]Note that the terms *low-fidelity* and *high-fidelity* prototypes are often used in the literature. We prefer the term *precision* because it refers to the content of the prototype itself, not its relationship to the final, as-yet-undefined system.

tices: a word processor designed for a professional typographer requires a different interaction design than one designed for secretaries, even though ostensibly they serve similar purposes. Designers must take the context of use into account when designing the details of the interaction (Beaudouin-Lafon, 2004).

A critical role for an interactive system prototype is to illustrate how the user will interact with the system. While this may seem more natural with online prototypes, in fact it is often easier to explore different interaction strategies with offline prototypes. Note that interactivity and precision are orthogonal dimensions. One can create an imprecise prototype that is highly interactive, such as a series of paper screen images in which one person acts as the user and the other plays the system. One may create a very precise but noninteractive prototype, such as a detailed animation that shows feedback from a specific action by a user.

Prototypes can support interaction in various ways. For offline prototypes, one person (often with help from others) plays the role of the interactive system, presenting information and responding to the actions of another person playing the role of the user. For online prototypes, parts of the software are implemented, while others are played by a person. (This approach, called the "Wizard of Oz" after the character in the 1939 movie of the same name, is explained in a later section.) The key is that the prototype feels interactive to the user.

Prototypes can support different levels of interaction. Fixed prototypes, such as video clips or precomputed animations, are noninteractive: the user cannot interact, or pretend to interact, with it. Fixed prototypes are often used to illustrate or test scenarios. Fixed-path prototypes support limited interaction. The extreme case is a fixed prototype in which each step is triggered by a prespecified user action. For example, the person controlling the prototype might present the user with a screen containing a menu. When the user points to the desired item, he or she presents the corresponding screen showing a dialogue box. When the user points to the word "OK," he or she presents the screen that shows the effect of the command. Even though the position of the click is irrelevant (it is used as a trigger), the person playing the role of the user gets the feel of the interaction. Of course, this type of prototype can be much more sophisticated, with multiple options at each step. Fixed-path prototypes are very effective with scenarios and can be used for horizontal and task-based prototypes (see section on prototyping strategies below).

Open prototypes support large sets of interactions. Such prototypes work like the real system, with some limitations. They usually cover only part of the system (see vertical prototypes) and often have limited error handling or reduced performance relative to that of the final system.

Prototypes may thus illustrate or test different levels of interactivity. Fixed prototypes simply illustrate what the interaction might look like. Fixed-path prototypes provide designers and users with the experience of what the interaction might feel like, but only in prespecified situations. Open prototypes allow designers and users to explore a wide range of possible forms of interaction with the system.

Evolution

Prototypes have different life spans: rapid prototypes are created for a specific purpose and then thrown away, iterative prototypes evolve, either to work out some details (increasing their precision) or to explore various alternatives, and evolutionary prototypes are designed to become part of the final system.

Rapid prototypes are especially important in the early stages of design. They must be inexpensive and easy to produce, since the goal is to quickly explore a wide variety of possible types of interaction and then throw them away. Note that rapid prototypes may be offline or online. Creating precise software prototypes, even if they must be reimplemented in the final version of the system, is important for detecting and fixing interaction problems. A later section presents specific prototyping techniques, both offline and online.

Iterative prototypes are developed as a reflection of a design in progress, with the explicit goal of evolving through several design iterations. Designing prototypes that support evolution is sometimes difficult. There is a tension between evolving toward the final solution and exploring an unexpected design direction, which may be adopted or thrown away completely. Each iteration should inform some aspect of the design. Some iterations explore different variations of the same theme. Others may systematically increase precision, working out the finer details of the interaction. A later section describes tools and techniques for creating iterative prototypes.

Evolutionary prototypes are a special case of iterative prototypes in which the prototype evolves into part or all of the final system (Fig. 52.1). Obviously, this only applies to software prototypes. Extreme Programming (Beck, 2000) advocates this approach, tightly coupling design and implementation and building the system through constant evolution of its components. Evolutionary prototypes require more planning and practice than the approaches above because the prototypes are both representations of the final system and the final system itself, making it more difficult to explore alternative designs. We advocate a combined approach, beginning with rapid prototypes and then using iterative or evolutionary prototypes according to the needs of the project. A later section describes how to create iterative and evolutionary prototypes, using appropriate development tools.

PROTOTYPES AND THE DESIGN PROCESS

In the previous section, we looked at prototypes as artifacts, i.e., the results of a design process. Prototypes can also be seen as artifacts for design, as an integral part of the design process. Prototyping helps designers think: prototypes are the tools they use to solve design problems. In this section, we focus on prototyping as a process and its relationship to the overall design process.

User-Centered Design

The field of HCI is both user centered (Norman & Draper, 1986) and iterative. User-centered design places the user at the center of the design process, from the initial analysis of user requirements to testing and evaluation. Prototypes support this goal by allowing users see and experience the final system long before it is built. Designers can identify functional requirements,

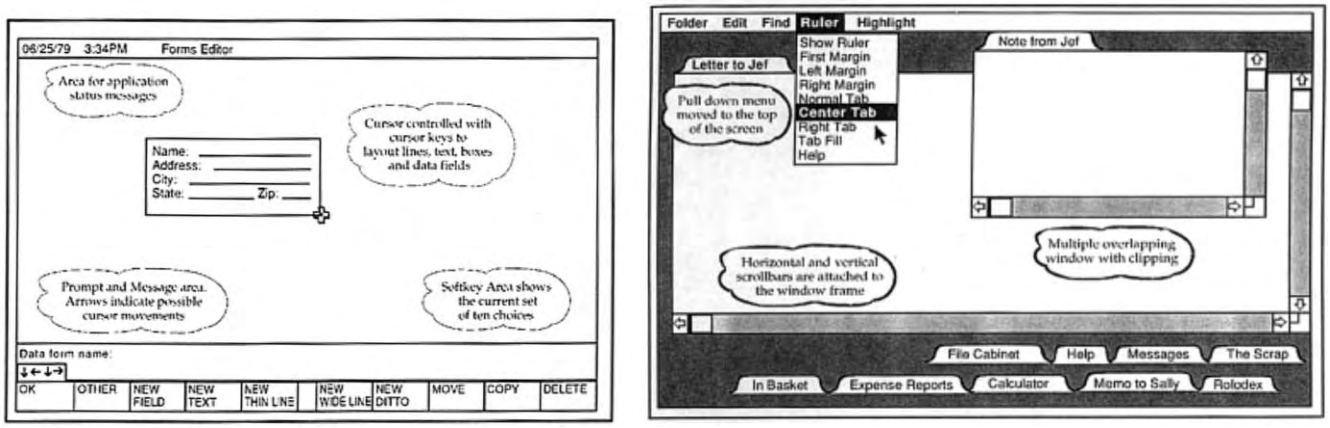

FIGURE 52.1. Evolutionary prototypes of the Apple Lisa: July 1979 (left), October 1980 (right) (Perkins et al., 1997) (© ACM, with permission).

usability problems and performance issues early and improve the design accordingly.

Iterative design involves multiple design-implement-test loops,[2] enabling the designer to generate different ideas and successively improve upon them. Prototypes support this goal by allowing designers to evaluate concrete representations of design ideas and select the best.

Prototypes reveal the strengths as well as the weaknesses of a design. Unlike pure ideas, abstract models or other representations, they can be contextualized to help understand how the real system would be used in a real setting. Because prototypes are concrete and detailed, designers can explore different real-world scenarios and users can evaluate them with respect to their current needs. Prototypes can be compared directly with other, existing systems, and designers can learn about the context of use and the work practices of the end users. Prototypes can help designers reanalyze the user's needs during the design process: not abstractly, as with traditional requirements analysis, but in the context of the system being built.

Participatory Design

Participatory design is a form of user-centered design that actively involves the user in all phases the design process (see Greenbaum & Kyng, 1991) Users are not simply consulted at the beginning and called in to evaluate the system at the end; they are treated as partners throughout. This early and active involvement of users helps designers avoid unpromising design paths and develop a deeper understanding of the actual design problem. Obtaining user feedback at each phase of the process also changes the nature of the final evaluation, which is used to fine tune the interface rather than discover major usability problems.

A common misconception about participatory design is that designers are expected to abdicate their responsibilities as designers, leaving the design to the end user. In fact, the goal is for designers and users to work together, each contributing their strengths to clarify the design problem as well as explore design solutions. Designers must understand what users can and cannot contribute. Usually, users are best at understanding the context in which the system will be used and subtle aspects of the problems that must be solved. Innovative ideas can come from both users and designers, but the designer is responsible for considering a wide range of options that might not be known to the user and balancing the trade-offs among them.

Because prototypes are shared, concrete artifacts, they serve as an effective medium for communication within the design team as well as with users. We have found that collaborating on prototype design is an effective way to involve users in participatory design. Prototypes help users articulate their needs and reflect on the efficacy of design solutions proposed by designers.

Exploring the Design Space

Design is not a natural science: the goal is not to describe and understand existing phenomena but to create something new. Of course, designers benefit from scientific research findings, and they may use scientific methods to evaluate interactive systems. However, designers also require specific techniques for generating new ideas and balancing complex sets of trade-offs to help them develop and refine design ideas.

Designers from fields such as architecture and graphic design have developed the concept of a design space, which constrains design possibilities along some dimensions, while leaving others open for creative exploration. Ideas for the design space come from many sources: existing systems, other designs, other designers, external inspiration, and accidents that prompt new ideas. Designers are responsible for creating a design space specific to a particular design problem. They explore this design space, expanding and contracting it as they add and eliminate ideas. The

[2]Software engineers refer to this as the Spiral model (Boehm, 1988).

process is iterative: more cyclic than reductionist. That is, the designer does not begin with a rough idea and successively add more details that are precise until the final solution is reached. Instead, he or she begins with a design problem, which imposes a set of constraints, and then generates a set of ideas to form the initial design space. He or she then explores this design space, preferably with the user, and selects a particular design direction to pursue. This closes off part of the design space, but opens up new dimensions that can be explored. The designer generates additional ideas along these dimensions, explores the expanded design space, and then makes new design choices. Design principles (e.g., Beaudouin-Lafon & Mackay, 2000) help this process by guiding it both in the exploration and choice phases. The process continues, in a cyclic expansion and contraction of the design space, until a satisfying solution is reached or time has run out.

All designers work with constraints: not just limited budgets and programming resources, but also design constraints. These are not necessarily bad: one cannot be creative along all dimensions at once. However, some constraints are unnecessary, derived from poor framing of the original design problem. If we consider a design space as a set of ideas and a set of constraints, the designer has two options. She can modify ideas within the specified constraints or modify the constraints to enable new sets of ideas. Unlike traditional engineering, which treats the design problem as a given, designers are encouraged to challenge, and if necessary, change the initial design problem. If she reaches an impasse, the designer can either generate new ideas or redefine the problem (and thus change the constraints). Some of the most effective design solutions derive from a more careful understanding and reframing of the design brief.

Note that all members of the design team, including users, may contribute ideas to the design space and help select design directions from within it. However, it is essential that these two activities are kept separate. Expanding the design space requires creativity and openness to new ideas. During this phase, everyone should avoid criticizing ideas and concentrate on generating as many as possible. Clever ideas, half-finished ideas, silly ideas, and impractical ideas all contribute to the richness of the design space and improve the quality of the final solution. In contrast, contracting the design space requires critical evaluation of ideas. During this phase, everyone should consider the constraints and weigh the trade-offs. Each major design decision must eliminate part of the design space: rejecting ideas is necessary in order to experiment and refine others and make progress in the design process. Choosing a particular design direction should spark new sets of ideas, and those new ideas are likely to pose new design problems. In summary, exploring a design space is the process of moving back and forth between creativity and choice.

Prototypes aid designers in both aspects of working with a design space: generating concrete representations of new ideas and clarifying specific design directions. The next two sections describe techniques that have proven most useful in our own prototyping work, both for research and product development.

Expanding the Design Space: Generating Ideas

The most well-known idea generation technique is brainstorming, introduced by Osborn (1957). His goal was to create synergy within the members of a group: ideas suggested by one participant would spark ideas in other participants. Subsequent studies (Collaros & Anderson, 1969; Diehl & Stroebe, 1987) challenged the effectiveness of group brainstorming, finding that aggregates of individuals could produce the same number of ideas as groups. They found certain effects, such as production blocking, free riding, and evaluation apprehension, were sufficient to outweigh the benefits of synergy in brainstorming groups. Since then, many researchers have explored different strategies for addressing these limitations. For our purpose, the quantity of ideas is not the only important measure: the relationships among members of the group are also important. As de Vreede et al. (2000) pointed out, one should also consider elaboration of ideas, as group members react to each other's ideas.

We have found that brainstorming, including a number of variants, is an important group-building exercise for participatory design. Of course, designers may brainstorm ideas by themselves. However, brainstorming in a group is more enjoyable and, if it is a recurring part of the design process, plays an important role in helping group members share and develop ideas together.

The simplest form of brainstorming involves a small group of people. The goal is to generate as many ideas as possible on a prespecified topic: quantity, not quality, is important. Brainstorming sessions have two phases: the first for generating ideas and the second for reflecting upon them. The initial phase should last no more than an hour. One person should moderate the session, keeping time, ensuring that everyone participates, and preventing people from critiquing each other's ideas. Discussion should be limited to clarifying the meaning of a particular idea. A second person records every idea, usually on a flipchart or transparency on an overhead projector. After a short break, participants are asked to reread all the ideas and each person marks their three favorite ideas.

One variation is designed to ensure that everyone contributes, not just those who are verbally dominant. Participants write their ideas on individual cards or Post-it notes for a prespecified period. The moderator then reads each idea aloud. Authors are encouraged to elaborate (but not justify) their ideas, which are then posted on a whiteboard or flipchart. Group members may continue to generate new ideas, inspired by the others they hear.

We use a variant of brainstorming, called "video brainstorming" (Mackay, 2000), as a very fast technique for prototyping interaction: instead of simply writing or drawing their ideas, participants act them out in front of a video camera (Fig. 52.2). The goal is the same as other brainstorming exercises, i.e., to create as many new ideas as possible, without critiquing them. However, the use of video, combined with paper or cardboard mock ups, encourages participants to experience the details of the interaction and to understand each idea from the perspective of the user, while preserving a tangible record of the idea.

Each video brainstorming idea should take two to five minutes to generate and capture, allowing participants to simulate a wide variety of ideas very quickly. The resulting video clips provide illustrations of each idea that are easier to understand (and remember) than hand-written notes. (We find that raw notes from brainstorming sessions are not very useful after a few weeks because the participants no longer remember the con-

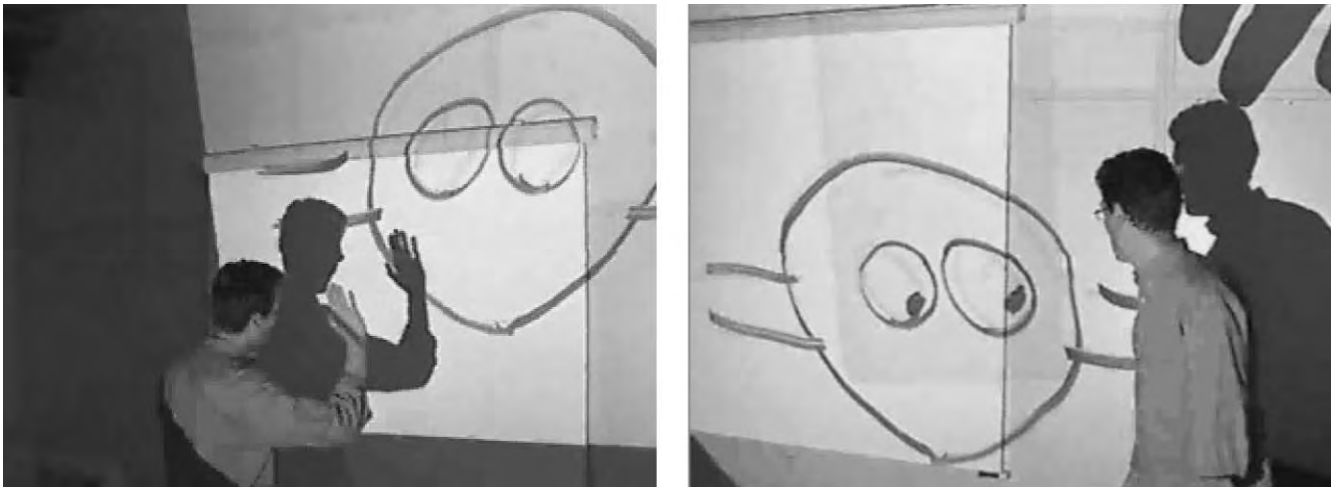

FIGURE 52.2. Video Brainstorming an animated character: One participant uses an overhead projector to project an image on the wall and responds to the actions of a second participant, who plays the role of the user. Here, the animated character, a very rough sketch on a transparency, reponds when the user waves and moves its eyes to follow the user.

text in which the ideas were created, whereas videobrainstorming clips are useful years later.)

Video brainstorming requires thinking more deeply about each idea than in traditional oral brainstorming. It is possible to stay vague and general when describing an interaction in words or even with a sketch, but acting out the interaction in front of the camera forces the author of the idea (and the other participants) to consider seriously the details of how a real user would actually interact with the idea. Video brainstorming also encourages designers and users to think about new ideas in the context in which they will be used. We also find that video clips from a video brainstorming session, even though rough, are much easier for the design team to interpret than written ideas from a standard brainstorming session.

We generally run a standard brainstorming session, either orally or with cards, prior to a video brainstorming session, in order to maximize the number of ideas to be explored. Participants then take their favorite ideas from the previous session and develop them further as video brainstorms. Each person is asked to direct at least two ideas, incorporating the hands or voices of other members of the group. We find that, unlike standard brainstorming, video brainstorming encourages even the quietest team members to participate.

Contracting the Design Space: Selecting Alternatives

After expanding the design space by creating new ideas, designers must stop and reflect upon the choices available to them. After exploring the design space, designers must evaluate their options and make concrete design decisions: choosing some ideas, specifically rejecting others, and leaving other aspects of the design open to further idea generation activities. Rejecting good, potentially effective ideas is difficult, but necessary to make progress.

Prototypes often make it easier to evaluate design ideas from the user's perspective. They provide concrete representations that can be compared. Many of the evaluation techniques described elsewhere in this handbook could be applied to prototypes, to help focus the design space. The simplest situation is when the designer must choose among several discrete, independent options. Running a simple experiment, using techniques borrowed from psychology, allows the designer to compare how users respond to each of the alternatives. The designer builds a prototype, with either fully implemented or simulated versions of each option. The next step is to construct tasks or activities that are typical of how the system would be used, and ask people from the user population to try each of the options under controlled conditions. It is important to keep everything the same, except for the options being tested.

Designers should base their evaluations on both quantitative measures, such as speed or error rate, and qualitative measures, such as the user's subjective impressions of each option. Ideally, of course, one design alternative will be clearly faster, prone to fewer errors, and preferred by the majority of users. More often, the results are ambiguous, and the designer must consider other factors when making the design choice. (Interestingly, running small experiments often highlights other design problems and may help the designer reformulate the design problem or change the design space.)

The more difficult (and common) situation, is when the designer faces a complex, interacting set of design alternatives, in which each design decision affects a number of others. Designers can use heuristic evaluation techniques, which rely on our understanding of human cognition, memory, and sensory-perception. They can also evaluate their designs with respect to ergonomic criteria or design principles (Beaudouin-Lafon & Mackay, 2000).

Another strategy is to create one or more scenarios that illustrate how the combined set of features will be used in a realistic

setting. The scenario must identify who is involved, where the activities take place, and what the user does over a specified period. Good scenarios involve more than a string of independent tasks; they should incorporate real-world activities, including common or repeated tasks, successful activities, breakdowns, and errors, with both typical and unusual events. The designer then creates a prototype that simulates or implements the aspects of the system necessary to illustrate each set of design alternatives. Such prototypes can be tested by asking users to walk through the same scenario several times, once for each design alternative. As with experiments and usability studies, designers can record both quantitative and qualitative data, depending on the level of the prototypes being tested.

The previous section described an idea-generation technique called "video brainstorming," which allows designers to generate a variety of ideas about how to interact with the future system. We call the corresponding technique for focusing in on a design "video prototyping." Video prototyping can incorporate any of the rapid-prototyping techniques (offline or online) described in a later section. Video prototypes are quick to build, force designers to consider the details of how users will react to the design in the context in which it will be used, and provide an inexpensive method of comparing complex sets of design decisions.

To an outside observer, video brainstorming and video prototyping techniques look very similar: both involve small design groups working together, creating rapid prototypes, and interacting with them in front of a video camera. Both result in video illustrations that make abstract ideas concrete and help team members communicate with each other. The critical difference is that video brainstorming expands the design space, by creating a number of unconnected collections of individual ideas, whereas video prototyping contracts the design space, by showing how a specific collection of design choices work together in a single design proposal.

Prototyping Strategies

Designers must decide what role prototypes should play with respect to the final system and in which order to create different aspects of the prototype. The next section presents four strategies: horizontal, vertical, task oriented, and scenario based, which focus on different design concerns. These strategies can use any of the prototyping techniques covered in the following sections.

Horizontal prototypes. The purpose of a horizontal prototype is to develop one entire layer of the design at the same time. This type of prototyping is most common with large software development teams, where designers with different skill sets address different layers of the software architecture. Horizontal prototypes of the user interface are useful to get an overall picture of the system from the user's perspective and address issues such as consistency (similar functions are accessible through similar user commands), coverage (all required functions are supported), and redundancy (the same function is/is not accessible through different user commands).

User interface horizontal prototypes can begin with rapid prototypes and progress through to working code. Software prototypes can be built with an interface builder without creating any of the underlying functionality, making it possible to test how the user will interact with the user interface without worrying about how the rest of the architecture works. However, some level of scaffolding or simulation of the rest of the application is often necessary; otherwise, the prototype cannot be evaluated properly. Consequently, software horizontal prototypes tend to be evolutionary, i.e., they are progressively transformed into the final system.

Vertical prototypes. The purpose of a vertical prototype is to ensure that the designer can implement the full, working system, from the user interface layer down to the underlying system layer. Vertical prototypes are often built to assess the feasibility of a feature described in a horizontal, task-oriented, or scenario-based prototype. For example, when we developed the notion of magnetic guidelines in the CPN2000 system to facilitate the alignment of graphical objects (Beaudouin-Lafon & Mackay, 2000, Beaudouin-Lafon & Lassen, 2000), we implemented a vertical prototype to test not only the interaction technique but also the layout algorithm and the performance. We knew that we could only include the particular interaction technique if we could implement a sufficiently fast response.

Vertical prototypes are generally high precision software prototypes because their goals are to validate ideas at the system level. They are often thrown away because they are generally created early in the project, before the overall architecture has been decided, and they focus on only one design question. For example, a vertical prototype of a spelling checker for a text editor does not require text-editing functions to be implemented and tested. However, the final version will need to be integrated into the rest of the system, which may involve considerable architectural or interface changes.

Task-oriented prototypes. Many user interface designers begin with a task analysis to identify the individual tasks that the user must accomplish with the system. Each task requires a corresponding set of functionality from the system. Task-based prototypes are organized as a series of tasks, which allows both designers and users to test each task independently, systematically working through the entire system.

Task-oriented prototypes include only the functions necessary to implement the specified set of tasks. They combine the breadth of horizontal prototypes, to cover the functions required by those tasks, with the depth of vertical prototypes, enabling detailed analysis of how the tasks can be supported. Depending on the goal of the prototype, both offline and online representations can be used for task-oriented prototypes.

Scenario-based prototypes. Scenario-based prototypes are similar to task-oriented ones, except that they do not stress individual, independent tasks, but rather follow a more realistic scenario of how the system would be used in a real-world setting. Scenarios are stories that describe a sequence of events and how the user reacts. A good scenario includes both common and unusual situations, and should explore patterns of activity over time. Bødker (1995) developed a checklist to ensure that no important issues have been left out.

We find it useful to begin with anecdotes derived from observations of or interviews with real users. Ideally, some of those users should participate in the creation of specific use and other users should critique them based on how realistic they are. Use scenarios are then turned into design scenarios, in which the same situations are described but with the functionality of the proposed system. Design scenarios are used, among other things, to create scenario-based video prototypes or software prototypes. Like task-based prototypes, the developer needs to write only the software necessary to illustrate the components of the design scenario. The goal is to create a situation in which the user can experience what the system would be like in a realistic use context, even if it addresses only a subset of the planned functionality.

The following section describes a variety of rapid prototyping techniques that can be used in any of these four prototyping strategies. We begin with offline rapid prototyping techniques, followed by online prototyping techniques.

RAPID PROTOTYPING

The goal of rapid prototyping is to develop prototypes very quickly, in a fraction of the time it would take to develop a working system. By shortening the prototype-evaluation cycle, the design team can evaluate more alternatives and iterate the design several times, improving the likelihood of finding a solution that successfully meets the user's needs. Rapid prototypes also serve to cut off unpromising design directions, saving time, and money. It is far easier to reject an idea based on a rapid prototype than a more fully developed one, and one reduces the chance of spending a great deal of time and effort on a design that ultimately does not work.

How rapid is rapid depends on the context of the particular project and the stage in the design process. Early prototypes, such as sketches, can be created in a few minutes. Later in the design cycle, a prototype produced in less than a week may still be considered rapid if the final system is expected to take months or years to build. Precision, interactivity, and evolution all affect the time it takes to create a prototype. Not surprisingly, a precise and interactive prototype takes more time to build than an imprecise or fixed one.

The techniques presented in this section are organized from most rapid to least rapid, according to the representation dimension previously introduced. Offline techniques are generally more rapid than online techniques. However, creating successive iterations of an online prototype may end up being faster than creating new offline prototypes.

Offline Rapid Prototyping Techniques

Offline prototyping techniques range from simple to very elaborate. Because they do not involve software, they are usually considered a tool for thinking through the design issues, to be thrown away when they are no longer needed. This section describes simple paper and pencil sketches, three-dimensional mock ups, Wizard of Oz simulations, and video prototypes.

Paper and pencil. The fastest form of prototyping involves paper, transparencies and Post-it notes to represent aspects of an interactive system (e.g., Muller, 1991). By playing the roles of both the user and the system, designers can get a quick idea of a wide variety of different layout and interaction alternatives, in a very short period of time (Rettig, 1994; Snyder, 2003).

Designers can create a variety of low-cost special effects. For example, a tiny triangle drawn at the end of a long strip cut from an overhead transparency makes a handy mouse pointer, which can be moved by a colleague in response to the user's actions. Post-it notes, with prepared lists, can provide pop-up menus. An overhead projector pointed at a whiteboard makes it easy to project transparencies (hand drawn or preprinted, overlaid on each other as necessary) to create an interactive display on the wall. The user can interact by pointing (Fig. 52.3) or drawing on the whiteboard. One or more people can watch the user and move the transparencies in response to her actions. Everyone in the room gets an immediate impression of how the eventual interface might look and feel.

Note that most paper prototypes begin with quick sketches on paper, then progress to more carefully drawn screen images made with a computer (Fig. 52.4). In the early stages, the goal is to generate a wide range of ideas and expand the design space, not to determine the final solution. Paper and pencil prototypes are an excellent starting point for horizontal, task-based, and scenario-based prototyping strategies.

Mock ups. Architects use mock ups or scaled prototypes to provide three-dimensional illustrations of future buildings. Mock ups are also useful for interactive system designers, helping them move beyond two-dimensional images drawn on paper or transparencies (see Bødker, Ehn, Knudsen, Kyng, & Madsen, 1988). Generally made of cardboard, foamcore, or other found materials (Frishberg, 2006), mock ups are physical prototypes of

FIGURE 52.3. Hand-drawn transparencies can be projected onto a wall, creating an interface a user can respond to.

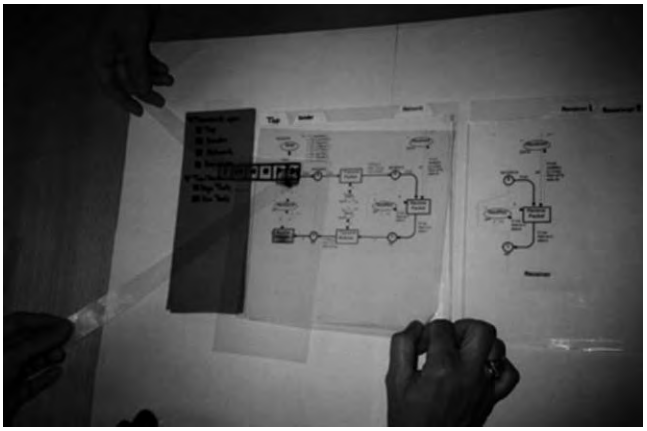

FIGURE 52.4. Several people work together to simulate interacting with this paper prototype. One person moves a transparency with a mouse pointer while another moves the diagram accordingly.

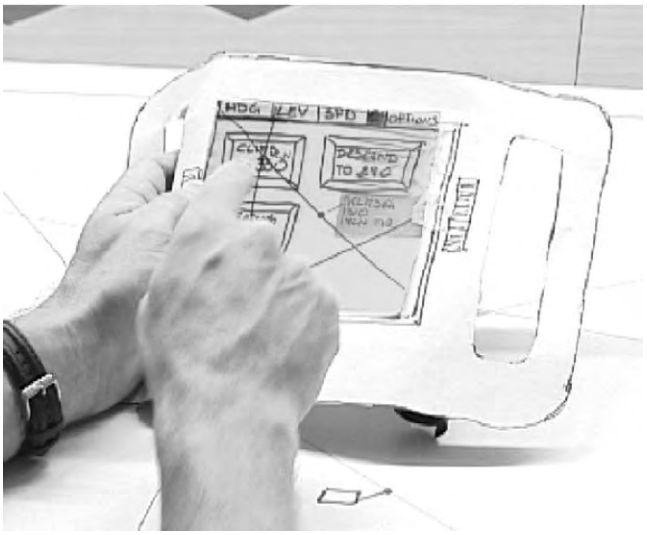

FIGURE 52.5 Mock-up of a hand-held display with carrying handle.

the new system. Fig. 52.5 shows an example of a handheld mock up showing the interface to a new handheld device. The mock up provides a deeper understanding of how the interaction will work in real-world situations than a set of screen images.

Mock ups allow the designer to concentrate on the physical design of the device, such as the position of buttons or the screen. The designer can also create several mock ups and compare input or output options, such as buttons versus trackballs. Designers and users should run through different scenarios, identifying potential problems with the interface or generating ideas for new functionality. Mock ups can also help the designer envision how an interactive system will be incorporated into a physical space (Fig. 52.6).

Wizard of Oz. Sometimes it is useful to give users the impression that they are working with a real system, even before it exists. Kelley (1983) dubbed this technique the *Wizard of Oz*,

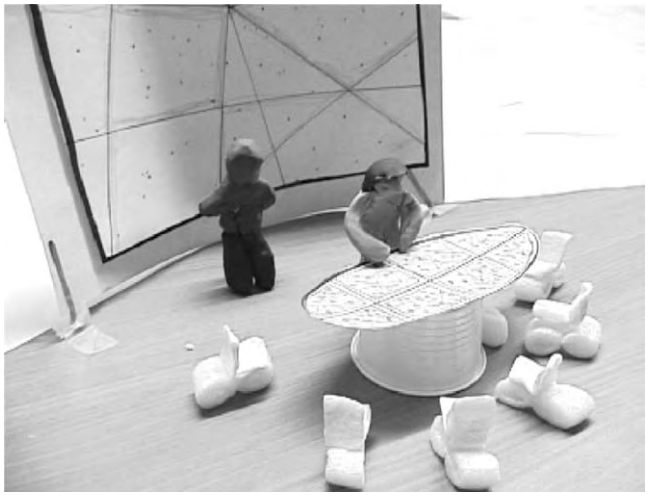

FIGURE 52.6. Scaled mock-up of an air traffic control table, connected to a wall display.

based on the scene in the 1939 movie of the same name. The heroine, Dorothy, and her companions ask the mysterious Wizard of Oz for help. When they enter the room, they see an enormous green human head breathing smoke and speaking with a deep, impressive voice. When they return later, they again see the Wizard. This time, Dorothy's small dog pulls back a curtain, revealing a frail old man pulling levers and making the mechanical Wizard of Oz speak. They realize that the impressive being before them is not a wizard at all, but simply an interactive illusion created by the old man.

The software version of the Wizard of Oz operates on the same principle. A user sits a screen and interacts with what appears to be a working program. Hidden elsewhere, the software designer (the Wizard) watches what the user does and, by responding to different user actions, creates the illusion of a working software program. In some cases, the user is unaware that a person, rather than a computer, is operating the system.

The Wizard of Oz technique lets users interact with partially functional computer systems. Whenever they encounter something that has not been implemented (or there is a bug), a human developer who is watching the interaction overrides the prototype system and plays the role destined to eventually be played by the computer. A combination of video and software can work well, depending upon what needs to be simulated.

The Wizard of Oz technique was initially used to develop natural language interfaces (e.g., Chapanis, 1982; Good et al., 1984). Since then, the technique has been used in a wide variety of situations, particularly those in which rapid responses from users are not critical. Wizard of Oz simulations may consist of paper prototypes, fully implemented systems, and everything in between.

Video prototyping. Video prototypes (Mackay, 1988) use video to illustrate how users will interact with the new system. As explained in an earlier section, they differ from video brainstorming in that the goal is to refine a single design, not generate new ideas. Video prototypes may build upon paper and pen-

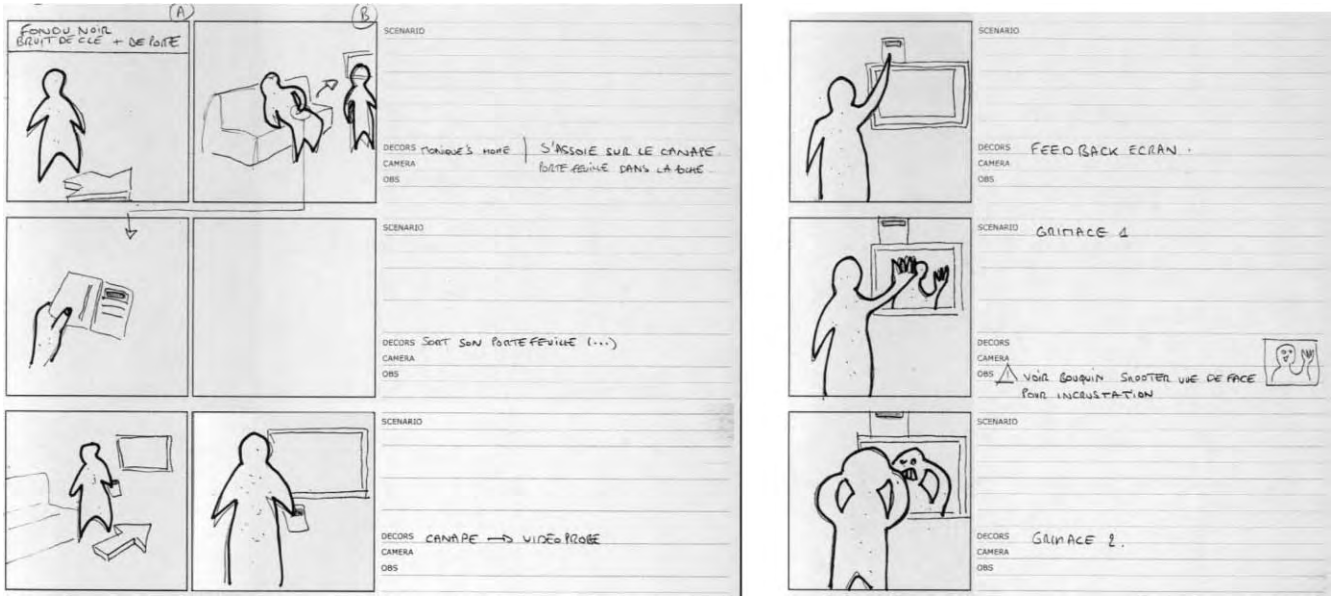

FIGURE 52.7. A storyboard for a tangible interface that enables users to establish and manage their connections to a small group of friends and family.

cil prototypes and cardboard mock ups and can use existing software and images of real-world settings.

We begin our video prototyping exercises by reviewing relevant data about users and their work practices, and then review ideas we video brainstormed. The next step is to create a use scenario, describing the user at work. Once the scenario is described in words, the designer develops a storyboard. Similar to a comic book, the storyboard shows a sequence of rough sketches of each action or event, with accompanying actions and/or dialogue (or subtitles), with related annotations that explain what is happening in the scene or the type of shot (Fig. 52.7). A paragraph of text in a scenario corresponds to about a page of a storyboard.

Storyboards help designers refine their ideas, generate "what- if" scenarios for different approaches to a story, and communicate with the other people who are involved in the design process. Some storyboards may be informal sketches of ideas, with only partial information. Others follow a predefined format and are used to direct the production and editing of a video prototype. Designers should jot down notes on storyboards as they think through the details of the interaction.

Paper storyboards can be used as is to communicate with other members of the design team. Designers and users can discuss the proposed system and alternative ideas for interacting with it (Fig. 52.8). Simple videos of each successive frame, with a voice over to explain what happens, can also be effective. We usually use storyboards to help us shoot video prototypes, which illustrate how a new system will look to a user in a real-world setting. We find that placing the elements of a storyboard on separate cards and arranging them (Mackay, & Pagani, 1994) helps the designer experiment with different linear sequences and insert or delete video clips. Continuing to the next step, i.e. creating a video prototype based on the storyboard, forces designers to consider the design details in even greater depth.

FIGURE 52.8 Video Prototyping: The CPN design team reviews their observations of CPN developers and then discuss several design alternatives. They work out a scenario and storyboard it, then shoot a video prototype that reflects their design.

Storyboards, even very informal ones, are essential guides for shooting video prototypes. To avoid spending time in post-production, we use a technique called "editing-in-the-camera" (see Mackay, 2000) in which each video clip, guided by the storyboard, is shot in the order that it will be viewed. With a well-designed storyboard, this is just as easy and results in a finished video prototype at the end of a one-hour session. Note that today's digital video cameras include editing features in the camera, which introduces the temptation to make editing changes on the fly. Our students who do this consistently take more time than their colleagues who do not, usually with worse results. In general, we recommend avoiding post hoc editing and just following the storyboard.

We use title cards, as in a silent movie, to separate the clips. This both simplifies shooting and makes it easier for the viewer to follow the story. Title cards also provide the only acceptable way to edit while you are shooting: if you make an error, you should address it immediately by rewinding to the last title card and continue shooting from there.

Video prototypes take several forms. In some, a narrator explains each event and several people on the sidelines may be necessary to move images and illustrate the interaction. In others, actors simply perform the movements and the viewer is expected to understand the interaction without a voice over. You can easily create simple special effects. For example, time-lapse photography allows you to have images appear and disappear based on the user's interaction. For example, record a clip of a user pressing a button, press "pause" on the camera, add a pop-up menu, then restart the camera, to create the illusion of immediate system feedback.

Video prototypes should begin with a title, followed by an establishing shot that shows the user in the context defined by the scenario. Next, create a series of close-up and midrange shots, interspersed with title cards to explain the story. Place a final title card with credits at the end. We print blank title cards on colored paper to make it easier to search for the sections of the video later: When you fast-forward through video, a solid blue or red frame clearly stands out (for detailed examples of video-based prototyping techniques, see Mackay, 2002).

Video prototypes are fixed prototypes. However, it is also possible to use video as an open prototype, in which users can interact with the prototype in an open-ended way. Video thus becomes a tool for sketching and visualizing interactions. For example, we sometimes use a live video camera as a Wizard of Oz tool and capture the interaction with a second video camera. The Wizard should have access to a set of prototyping materials representing screen objects. Other team members stand by ready to help move objects as needed. The live camera is pointed at the Wizard's work area, with either a paper prototype or a partially working software simulation. The resulting image is projected onto a screen or monitor in front of the user. One or more people should be situated so that they can observe the actions of the user and manipulate the projected video image accordingly. This is most effective if the Wizard is well prepared for a variety of events and can present semiautomated information. The user interacts with the objects he or she sees on the screen and the Wizard moves the relevant materials in direct response to each user action. The other camera records the interaction between the user and the simulated software system on the screen, to create either a video brainstorm (for a quick idea) or a fully storyboarded video prototype.

Fig. 52.9 shows a Wizard of Oz simulation with a live video camera, video projector, whiteboard, overhead projector, and transparencies. The setup allows two people to experience how they would communicate via a new interactive communication system. One video camera films the blond woman, who can see and talk to the brunette woman. Her image is projected live onto the left side of the wall. An overhead projector displays hand drawn transparencies, manipulated by two other people, in response to gestures made by the brunette woman. The entire interaction is videotaped by a second video camera. Note that participants at a workshop on user interfaces for air-traffic control created this video: none of the participants had ever used video prototyping techniques but they were able to set up this Wizard of Oz style environment and use it to generate new interaction ideas in less than 30 minutes.

FIGURE 52.9. Complex wizard-of-Oz simulation, with projected image from a live video camera and transparencies projected from an overhead projector.

Combining Wizard of Oz and video is a particularly powerful prototyping technique because it gives the person playing the user a real sense of what it might actually feel like to interact with the proposed tool, long before it has been implemented. Seeing a video clip of someone else interacting with a simulated tool is more effective than simply hearing about it, but interacting with it directly is more powerful still. Note that video should be used with caution, particularly when video prototypes are taken out of their original design setting (for a more detailed discussion of the ethical issues involved, see Mackay, 1995).

Video prototyping may act as a form of specification for developers, enabling them to build the precise interface, both visually and interactively, created by the design team. This is particularly useful when moving from offline to online prototypes, which we now describe.

Online Rapid Prototyping Techniques

The goal of online rapid prototyping is to create higher precision prototypes than can be achieved with offline techniques. Such prototypes may prove useful to better communicate ideas to clients, managers, developers and end users. They are also useful for the design team to fine tune the details of a layout or an interaction. They may exhibit problems in the design that were not apparent in less precise prototypes. Finally, they may be used early on in the design process for low precision prototypes that would be difficult to create offline, such as when very dynamic interactions or visualizations are needed.

The techniques presented in this section are sorted by interactivity. We start with noninteractive simulations, such as animations, followed by interactive simulations that provide fixed or multiple-paths interactions. We finish with scripting languages, which support open interactions.

Noninteractive simulations. A noninteractive simulation is a computer-generated animation that represents what a person would see of the system if he or she were watching over the user's shoulder. Noninteractive simulations are usually created when offline prototypes, including video, fail to capture a particular aspect of the interaction and it is important to have a quick prototype to evaluate the idea. It is usually best to start by creating a storyboard to describe the animation, especially if the developer of the prototype is not a member of the design team.

One of the most widely used tools for noninteractive simulations is Macromedia Director™. The designer defines graphical objects called "sprites," and defines paths along which to animate them. The succession of events, such as when sprites appear and disappear, is determined with a time line. Sprites are usually created with drawing tools, such as Adobe Illustrator or Deneba Canvas, painting tools, such as Adobe Photoshop, or even scanned images. Director is a very powerful tool and experienced developers use it quickly to create sophisticated interactive simulations. (However, it is still faster to create noninteractive simulations.) Other similar tools exist on the market such as Adobe AfterEffects and Macromedia Flash (Fig. 52.10).

Fig. 52.11 shows a set of animation movies created by Curbow to explore the notion of accountability in computer sys-

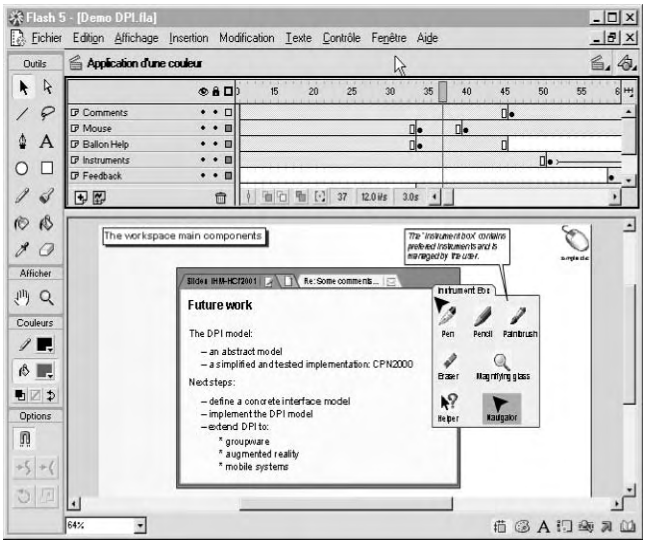

FIGURE 52.10. A non-interactive simulation of a desktop interface created with Macromedia Flash. The time-line (top) displays the active sprites while the main window (bottom) shows the animation. (© O. Beaudoux, with permission)

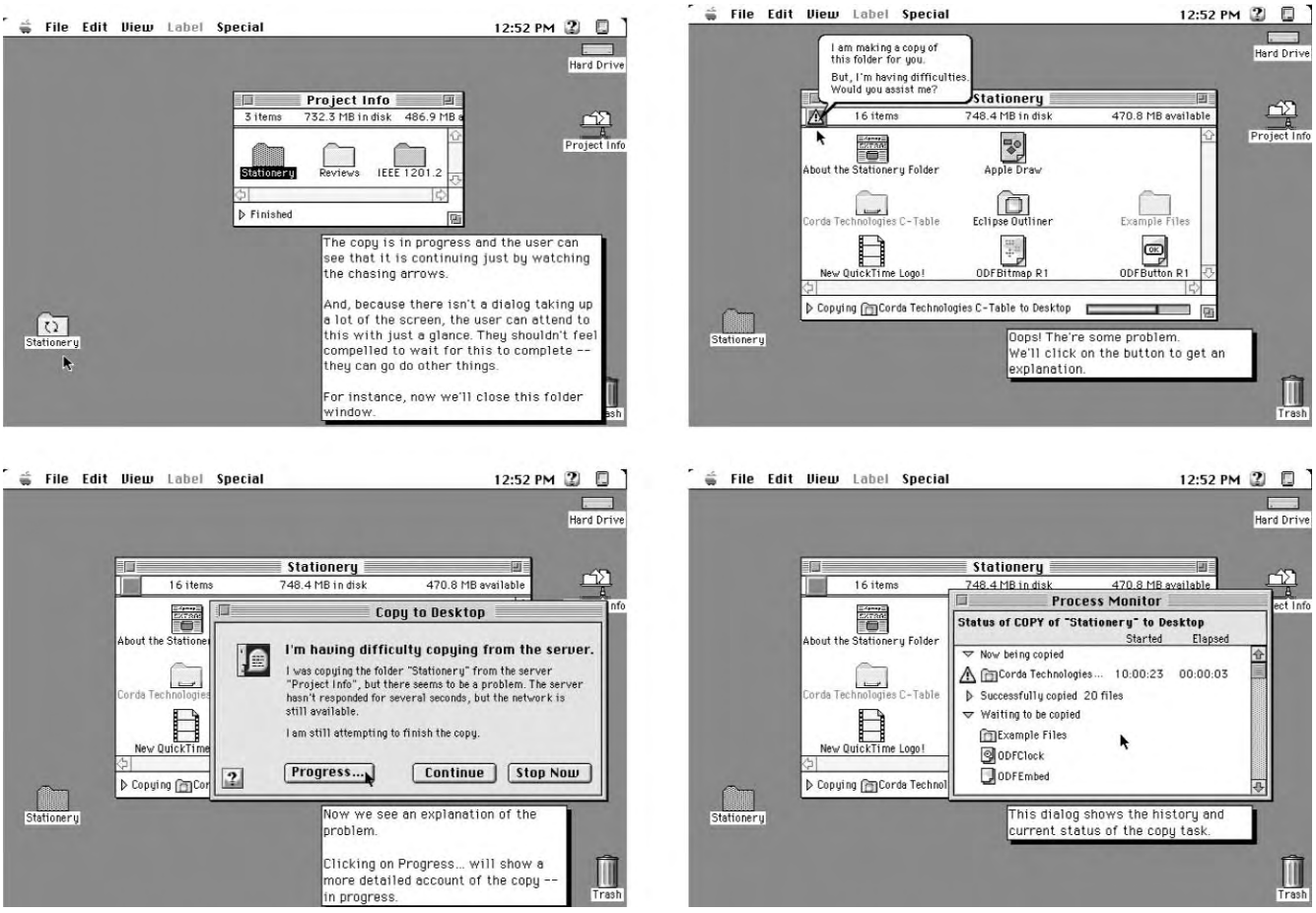

FIGURE 52.11. Frames from an animated simulation created with Macromind Director (© D. Curbow, with permission)

tems (Dourish, 1997). These prototypes explore new ways to inform the user of the progress of a file copy operation. They were created with Macromind Director by combining custom-made sprites with sprites extracted from snapshots of the Macintosh Finder. The simulation features cursor motion, icons being dragged, windows opening and closing, and so forth. The result is a realistic prototype that shows how the interface looks and behaves, that was created in just a few hours. Note that the simulation also features text annotations to explain each step, which helps document the prototype.

Noninteractive animations can be created with any tool that generates images. For example, many web designers use Adobe Photoshop to create simulations of their websites. Photoshop images are composed of various layers that overlap like transparencies. The visibility and relative position of each layer can be controlled independently. Designers can quickly add or delete visual elements, simply by changing the characteristics of the relevant layer. This permits quick comparisons of alternative designs and helps visualize multiple pages that share a common layout or banner. Skilled Photoshop users find this approach much faster than most web-authoring tools.

We used this technique in the CPN2000 project (Mackay, Ratzer, & Janecek, 2000) to prototype the use of transparency. After several prototyping sessions with transparencies and overhead projectors, we moved to the computer to understand the differences between the physical transparencies and the transparent effect as it would be rendered on a computer screen. We later developed an interactive prototype with OpenGL, which required an order of magnitude more time to implement than the Photoshop mock up.

Even a spreadsheet program can be used for prototyping: Berger (2006) described the use of Microsoft Excel to prototype form-based interfaces. First, a template is created that contains a number of reusable elements by taking advantage of the workbook feature of Excel, where multiple pages can be presented with a tabbed interface. Then, prototypes are created by copying and pasting items from the template into a blank page, taking advantage of the table structure of the spreadsheet to create grid layouts.

Interactive simulations. Designers can also use tools such as Adobe Photoshop to create Wizard of Oz simulations. For example, the effect of dragging an icon with the mouse can be obtained by placing the icon of a file in one layer and the icon of the cursor in another layer and by moving either or both layers. The visibility of layers, as well as other attributes, can also create more complex effects. Like Wizard of Oz and other paper prototyping techniques, the behavior of the interface is generated by the user who is operating the Photoshop interface.

More specialized tools, such as Hypercard and Macromedia Director, can be used to create simulations with which the user can directly interact. Hypercard (Goodman, 1987) was one of the most successful early prototyping tools. It was an authoring environment based on a stack metaphor: a stack contains a set of cards that share a background, including fields and buttons. Each card can also have its own unique contents, including fields and buttons (Fig. 52.12). Using a scripting language, stacks, cards, fields, and buttons can react to user events, such as clicking a button, as well as system events, for example, when a new card is displayed or about to disappear.

Interfaces can be prototyped quickly with this approach, by drawing different states in successive cards and using buttons to switch from one card to the next. Multiple-path interactions can be programmed by using several buttons on each card. Interactions that are more open require advanced use of the scripting language, but are easy to master with a little practice.

Macromind Director uses a different metaphor, attaching behaviors to sprites and to frames of the animation. For example, a button can be defined by attaching a behavior to the sprite representing that button. When the sprite is clicked, the animation jumps to a different sequence. This is usually coupled with a behavior attached to the frame containing the button that loops the animation on the same frame. As a result, nothing happens until the user clicks the button, at which point the animation skips to a sequence where, for example, a dialogue box opens. The same technique can be used to make the OK and Cancel buttons of the dialogue box interactive. Typically, the Cancel button would skip to the original frame while the OK button would skip to a third sequence. Director comes with a large library of behaviors to describe such interactions, so that prototypes can be created completely interactively. New behaviors can also be defined with a scripting language called Lingo.

Scripting languages. Scripting languages are the most advanced rapid prototyping tools. As with the interactive-simulation tools described above, the distinction between rapid prototyping tools and development tools is not always clear. Scripting languages make it easy to quickly develop throwaway prototypes (a few hours to a few days), which may or may not be used in the final system, for performance or other technical reasons.

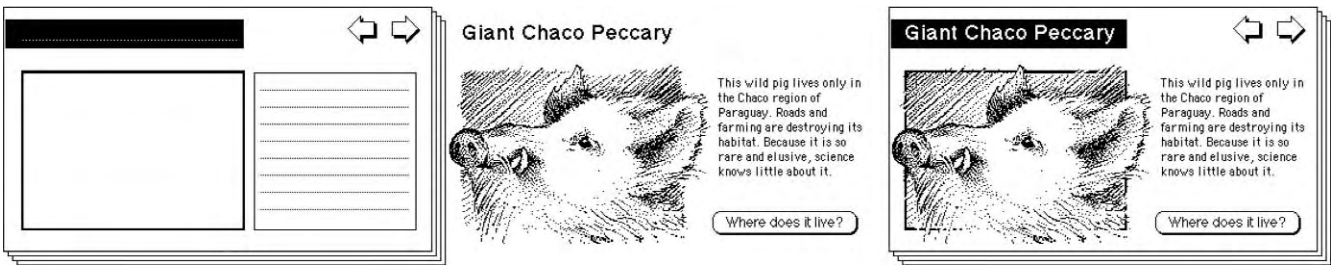

FIGURE 52.12. A Hypercard card (right) is the combination of a background (left) and the card's content (center) (© Apple Computer, with permission)

A scripting language is a programming language that is both lightweight and easy to learn. Most scripting languages are interpreted or semicompiled; for example, the user does not need to go through a compile-link-run cycle each time the script (program) is changed. Scripting languages can be forbidding: they are not strongly typed and nonfatal errors are ignored unless explicitly trapped by the programmer. Scripting languages are often used to write small applications for specific purposes and can serve as glue between preexisting applications or software components.

Tcl (Ousterhout, 1993) is particularly suitable to develop user interface prototypes (or small to medium-size applications) because of its Tk user interface toolkit. Tcl was inspired by the syntax of the Unix shell and makes it very easy to interface existing applications by turning the application programming interface (API) into a set of commands that can be called directly from a Tcl script. Tk features all the traditional interactive objects (called "widgets") of a UI toolkit: buttons, menus, scrollbars, lists, dialogue boxes, and so forth. A widget is typically created with only one line. For example,

```
button .dialogbox.ok -text OK -command {de-
stroy .dialogbox}.
```

This command creates a button, called ".dialogbox.ok," whose label is "OK." It deletes its parent window ".dialogbox" when the button pressed. A traditional programming language and toolkit would take 5 to 20 lines to create the same button.

Tcl also has two advanced, heavily parameterized widgets: the text widget and the canvas widget. The text widget can be used to prototype text-based interfaces. Any character in the text can react to user input using tags. For example, it is possible to turn a string of characters into a hypertext link. In Beaudouin-Lafon (2000), the text widget was used to prototype a new method for finding and replacing text. When entering the search string, all occurrences of the string are highlighted in the text (Fig. 52.13). Once a replace string has been entered, clicking an occurrence replaces it (the highlighting changes from yellow to red). Clicking a replaced occurrence returns it to its original value. This example also uses the canvas widget to create a custom scrollbar that displays the positions and status of the occurrences.

The Tk canvas widget is a drawing surface that can contain arbitrary objects: lines, rectangles, ovals, polygons, text strings, and widgets. Tags allow behaviors (e.g., scripts) that are called when the user acts on these objects. For example, an object that can be dragged will be assigned a tag with three behaviors: button-press, mouse-move, and button-up. Because of the flexibility of the canvas, advanced visualization and interaction techniques can be implemented more quickly and easily than with other tools. For example, Fig. 52.14 shows a prototype exploring new ideas to manage overlapping windows on the screen (Beaudouin-Lafon, 2001). Windows can be stacked and slightly rotated so that it is easier to recognize them, and they can be folded so it is possible to see what is underneath without having to move the window. Even though the prototype is not perfect e.g., folding a window that contains text is not properly supported, it was instrumental in identifying a number of problems with the interaction techniques and finding appropriate solutions through iterative design.

Tcl and Tk can also be used with other programming languages. For example, Pad++ (Bederson & Meyer, 1998) is implemented as an extension to Tcl/Tk: the zoomable interface is implemented in C for performance and is accessible from Tk as a new widget. This makes it easy to prototype interfaces that use zooming. It is also a way to develop evolutionary prototypes: a first prototype is implemented completely in Tcl, then parts of it are reimplemented in a compiled language to performance. Ultimately, the complete system may be implemented in another language, although it is more likely that some parts will remain in Tcl.

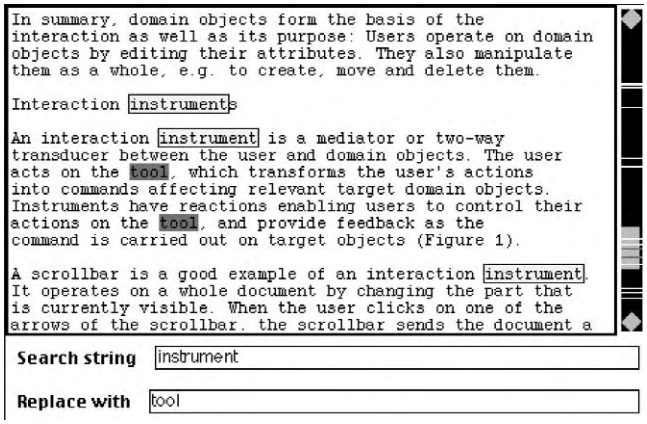

FIGURE 52.13. Using the Tk text and canvas widgets to prototype a novel search and replace interaction technique (Beaudouin-Lafon, 2000).

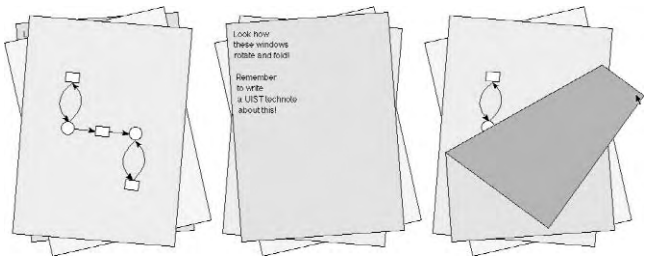

FIGURE 52.14. Using the Tk canvas widget to prototype a novel window manager (Beaudouin-Lafon, 2001).

Software prototypes can also be used in conjunction with hardware prototypes. Fig. 52.15 shows an example of a hardware prototype that captures handwritten text from a paper flight strip (using a combination of a graphics tablet and a custom-designed system for detecting the position of the paper strip holder). We used Tcl/Tk, in conjunction with C++, to present information on a RADAR screen (tied to an existing air traffic control simulator) and to provide feedback on a touch-sensitive display next to the paper flight strips (Caméléon, Mackay, et al., 1998). The user can write in the ordinary way on the paper flight strip, and the system interprets the gestures according to the location of the writing on the strip. For example, a change in flight level is automatically sent to another controller for confirmation and a physical tap on the strip's ID lights up the corresponding aircraft on the RADAR screen. A later section of this chapter expands this approach to the development of prototypes of mixed reality and pervasive computing systems.

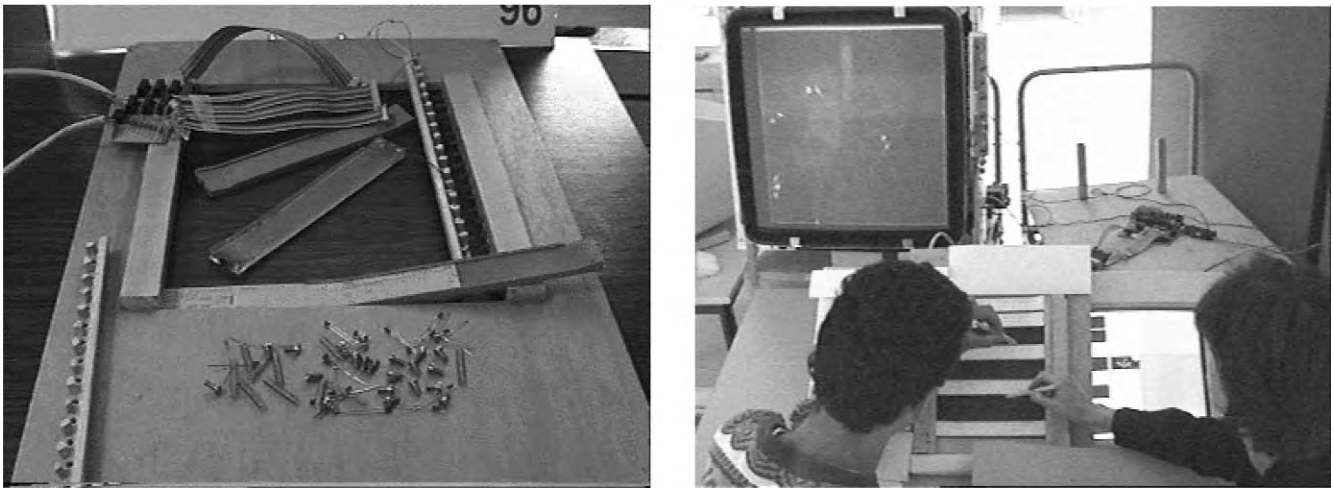

FIGURE 52.15. Caméléon's augmented stripboard (left) is a working hardware prototype that identifies and captures hand-writing from paper flight strips. Members of the design team test the system (right), which combines both hardware and software prototypes into a single interactive simulation.

ITERATIVE AND EVOLUTIONARY PROTOTYPES

Prototypes may also be developed with traditional software development tools. In particular, high-precision prototypes usually require a level of performance that cannot be achieved with the rapid online prototyping techniques previously described. Similarly, evolutionary prototypes intended to evolve into the final product require more traditional software development tools. Finally, even shipped products are not final, since subsequent releases can be viewed as initial designs for prototyping the next release.

Development tools for interactive systems have been in use for over 20 years and are constantly being refined. Several studies have shown that the part of the development cost of an application spent on the user interface is 50% to 80% of the total cost of the project (Myers & Rosson, 1992). The goal of development tools is to shift this balance by reducing production and maintenance costs. Another goal of development tools is to anticipate the evolution of the system over successive releases and support iterative design.

The lowest level tools are graphical libraries, which provide hardware independence for painting pixels on a screen and handling user input and window systems that provide an abstraction (the window) to structure the screen into several virtual terminals. User interface toolkits structure an interface as a tree of interactive objects called "widgets," while user interface builders provide an interactive application to create and edit those widget trees. Application frameworks build on toolkits and UI builders to facilitate creation of typical functions such as cut/copy/paste, undo, help and interfaces based on editing multiple documents in separate windows. Model-based tools semiautomatically derive an interface from a specification of the domain objects and functions to be supported. Finally, user interface development environments (UIDEs) provide an integrated collection of tools for the development of interactive software.

Before we describe some of these categories in more detail, it is important to understand how they can be used for prototyping. It is not always best to use the highest level available tool for prototyping. High-level tools are most valuable in the long term because they make it easier to maintain the system, port it to various platforms, or localize it to different languages. These issues are irrelevant for vertical and throwaway prototypes, so a high-level tool may prove less effective than a lower level one.

The main disadvantage of higher level tools is that they constrain or stereotype the types of interfaces they can implement. User interface toolkits usually contain a limited set of widgets, and it is expensive to create new ones. If the design must incorporate new interaction techniques, such as bimanual interaction (Kurtenbach, Fitzmaurice, Baudel, & Buxton, 1997) or zoomable interfaces (Bederson & Hollan, 1994), a user interface toolkit will hinder rather than help prototype development. Similarly, application frameworks assume a stereotyped application with a menu bar, several toolbars, a set of windows holding documents, and so forth. Such a framework would be inappropriate for developing a game or a multimedia educational CD-ROM that requires a fluid, dynamic, and original user interface.

Finally, developers need to truly master these tools, especially when prototyping in support of a design team. Success depends on the programmer's ability to change the details quickly as well as the overall structure of the prototype. A developer will be more productive when using a familiar tool than if forced to use a more powerful but unknown tool.

Since a complete tour of development tools for interactive systems is beyond the scope of this chapter, we focus on those tools that can be used most effectively for prototyping: user interface toolkits, user interface builders, and user interface development environments.

User Interface Toolkits

User interface toolkits are probably the most widely used tool nowadays to implement applications. All three major platforms (Unix/Linux, MacOS, and Windows) come with at least one standard UI toolkit. The main abstraction provided by a UI toolkit is the widget. A widget is a software object that has three facets that closely match the MVC model (Krasner & Pope, 1988): a presentation, a behavior, and an application interface.

The presentation defines the graphical aspect of the widget. The overall presentation of an interface is created by assembling widgets into a tree. Widgets such as buttons are the leaves of the tree. Composite widgets constitute the nodes of the tree and control the layout of their children. The behavior of a widget defines the interaction methods it supports: a button can be pressed, a scrollbar can be scrolled, and a text field can be edited. The application interface defines how a widget communicates the results of the user interaction to the rest of the application. It is usually based on a notification mechanism.

One limitation of widgets is that their behaviors are limited to the widget itself. Interaction techniques that involve multiple widgets, such as drag-and-drop, cannot be supported by the widgets' behaviors alone and require separate support in the UI toolkit. Some interaction techniques, such as toolglasses or magic lenses (Bier, Stone, Pier, Buxton, & De Rose, 1993), break the widget model both with respect to the presentation and the behavior and cannot be supported by traditional toolkits. In general, prototyping new interaction techniques requires either implementing them within new widget classes, which is not always possible, or not using a toolkit at all. Implementing a new widget class is typically more complicated than implementing the new technique outside the toolkit, for example, with a graphical library, and is rarely justified for prototyping. Many toolkits provide a blank widget, such as the Canvas in Tk or JFrame in Java Swing (Eckstein, Loy, & Wood, 1998), which can be used by the application to implement its own presentation and behavior. This is usually a good alternative to implementing a new widget class, even for production code.

User interface toolkits have been an active area of research over the past 15 years. InterViews (Linton, Vlissides, & Calder, 1989) inspired many modern toolkits and user interface builders. Some recent research toolkits that can be used for prototyping include SubArctic (Hudson, Mankoff, & Smith, 2005) and Satin (Hong & Landay, 2000). The latter is dedicated to ink-based interaction and is used with the Silk and Denim UIDEs described below.

A number of toolkits have also shifted away from the widget model to address other aspects of user interaction. For example, GroupKit (Roseman & Greenberg, 1996, 1999) was designed for groupware, Jazz (Bederson, Meyer, & Good, 2000) for zoomable interfaces, the Visualization (Schroeder, Martin, & Lorensen, 1997) and InfoVis (Fekete, 2004) toolkits for visualization, Inventor (Strass, 1993) for 3-D graphics, and Metisse (Chapuis & Roussel, 2005) for window management (Fig. 52.16).

Creating an application or a prototype with a UI toolkit requires solid knowledge of the toolkit and experience with programming interactive applications. In order to control the complexity of the interrelations between independent pieces of code (creation of widgets, callbacks, global variables, etc.), it is important to use well-known design patterns (Gamma, Helm, Johnson, & Vlissides, 1995) such as Command, Chain of responsibility, Mediator, and Observer. Otherwise the code quickly becomes unmanageable and, in the case of a prototype, unsuitable to design space exploration.

User Interface Builders

User interface builders leverage user interface toolkits by allowing the developer of an interactive system to create the presentation of the user interface, such as the tree of widgets, interactively with a graphical editor. The editor typically features a

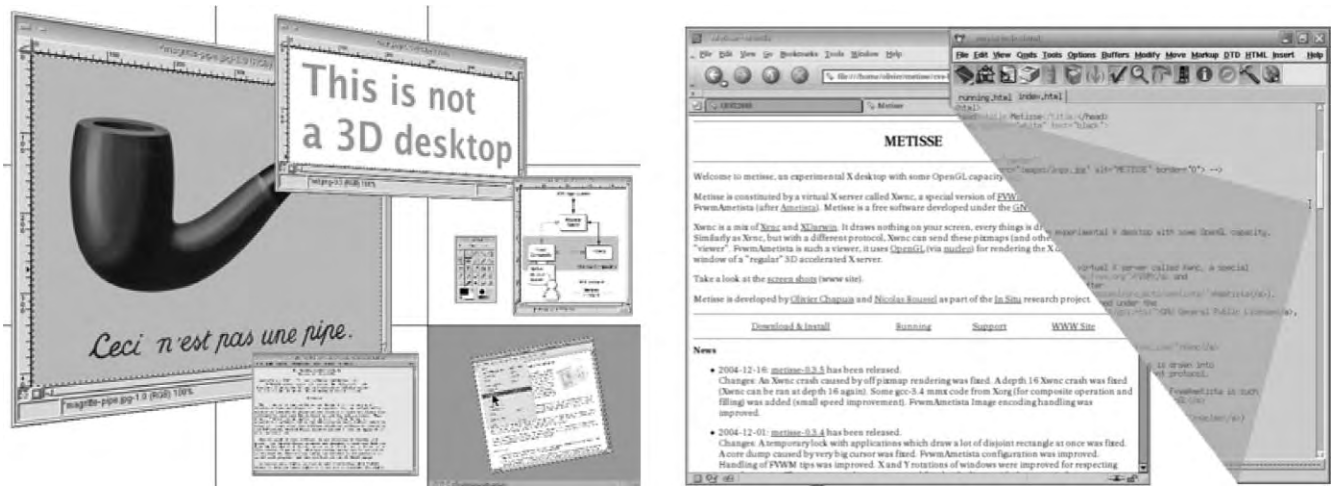

FIGURE 52.16. Two prototypes of a window system implemented with Metisse. The one on the right implements folding windows. Unlike in Figure 15 above, here it works with real applications. (© O. Chapuis, with permission)

palette of widgets that the user can use to draw the interface in the same way as a graphical editor is used to create diagrams with lines, circles, and rectangles. The presentation attributes of each widget can be edited interactively as well as the overall layout. This saves a lot of time that would otherwise be spent writing and fine tuning rather dull code that creates widgets and specifies their attributes. It also makes it extremely easy to explore and test design alternatives.

Apple's Interface Builder (Fig. 52.17) is a descendant of the NeXT interface builder (NeXT Corporation, 1991). The palette to the right contains the available widgets. The user interface of the application was created by dragging these widgets into the application window at the top left. The bottom left window contains icons representing the application objects. By dragging connectors between widgets and these objects, a significant part of the behavior of the interface can be created interactively. The user interface can be tested at any time by switching the builder to test mode, making it easy to verify that it behaves as expected. The same application built directly with the underlying toolkit would require dozens of lines of code and significant debugging.

User interface builders are widely used to develop prototypes, as well as final applications. They are easy to use, they make it easy to change the look of the interface, and they hide a lot of the complexity of creating user interfaces with UI toolkits. However, despite their name, they do not cover the whole user interface. Therefore they still require a significant amount of programming, a good knowledge of the underlying toolkit and an understanding of their limits, especially when prototyping novel visualization and interaction techniques.

User Interface Development Environments

A number of high-level tools exist for creating interactive applications. They are often based on user interface toolkits, and they sometimes include an interface builder. These tools are often referred to as "user interface development environments."

The simplest of these tools are application frameworks, which address stereotyped applications. For example, many applications have a standard form where windows represent documents that can be edited with menu commands and tools from palettes; each document may be saved into a disk file; standard functions such as copy/paste, undo, and help are supported. Implementing such stereotyped applications with a UI toolkit or UI builder requires replicating a significant amount of code to implement the general logic of the application and the basics of the standard functions. Application frameworks address this issue by providing a shell that the developer fills in with the functional core and the actual presentation of the nonstandard parts of the interface. Most frameworks have been inspired by MacApp, a framework developed in the 1980s to develop applications for the Macintosh (Apple Computer, 1996). Some frameworks are more specialized than MacApp. For example, Unidraw (Vlissides & Linton, 1990) is a framework for creating graphical editors in domains, such as technical and artistic drawing, music composition, or circuit design. By addressing a smaller set of applications, such a framework can provide more support and significantly reduce implementation time.

Mastering an application framework takes time. It requires knowledge of the underlying toolkit and the design patterns used in the framework, and a good understanding of the de-

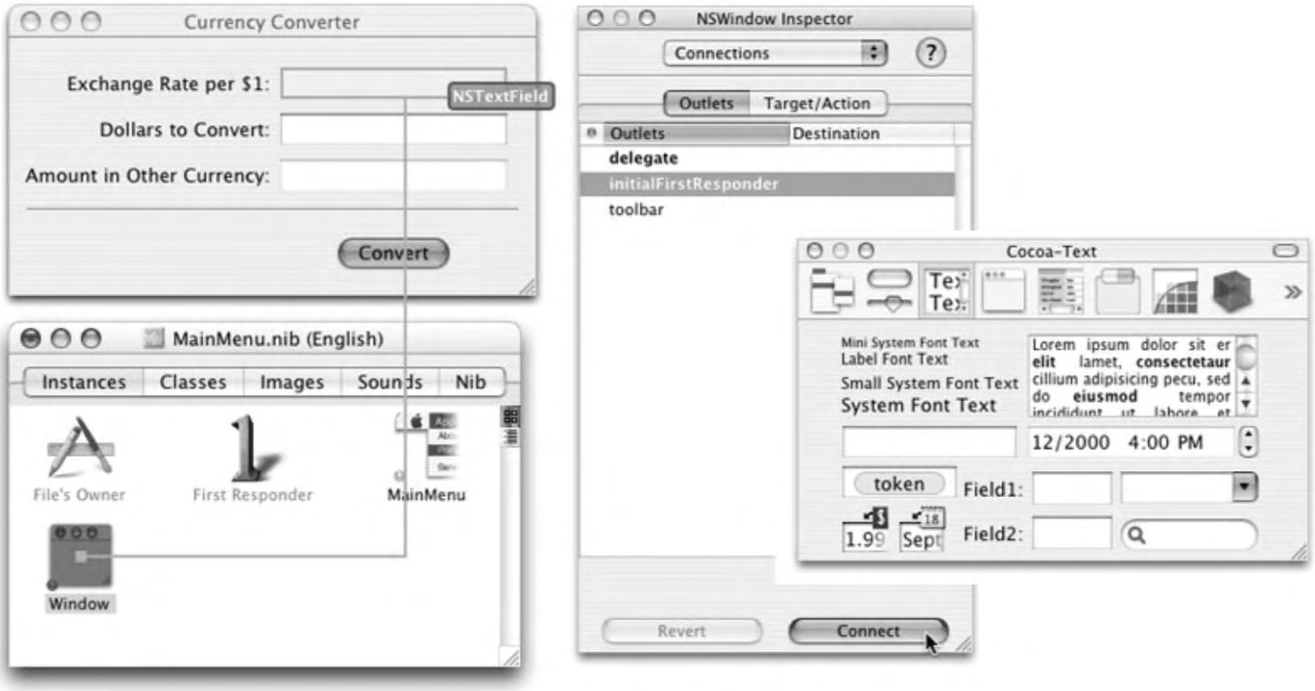

FIGURE 52.17. Interface Builder with the window being built (top-left), application objects (bottom-left), inspector (center) and widget palette (right).

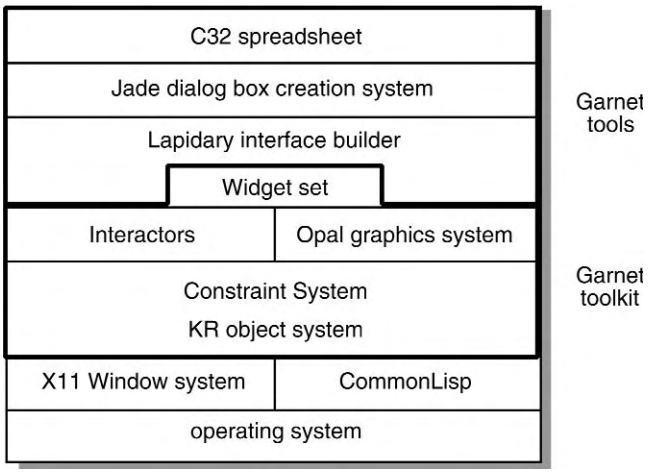

FIGURE 52.18. The Garnet toolkit and tools (Myers et al., 1990).

sign philosophy of the framework. A framework is useful because it provides a number of functions "for free," but at the same time it constrains the design space that can be explored. Frameworks can prove effective for prototyping if their limits are well understood by the design team.

UIDEs consist in assembling a set of tools into an environment where different aspects of an interactive system can be specified and generated separately. For example, Garnet (Fig. 52.18) and its successor Amulet (Myers, Giuse, et al., 1990; Myers, McDaniel, et al., 1997) provide a comprehensive set of tools, including a traditional user interface builder, a semiautomatic tool for generating dialogue boxes, a user interface builder based on a demonstration approach, and so forth.

Some UIDEs include tools that are specifically designed for prototyping. For example, Silk (Landay & Myers, 2001) is a tool aimed at the early stages of design, when interfaces are sketched rather than prototyped in software. The user sketches a user interface directly on the screen (Fig. 52.19). Using gesture

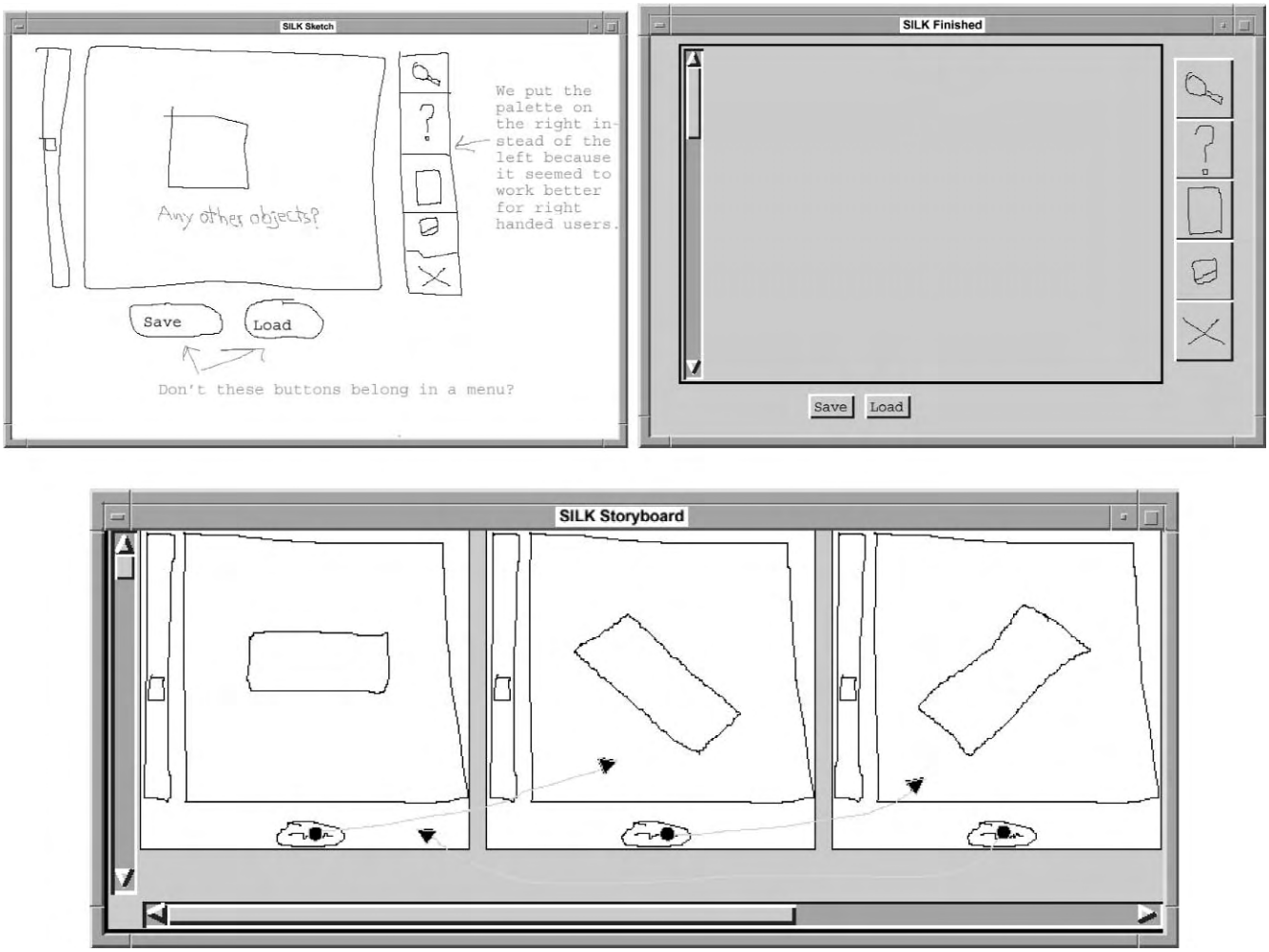

FIGURE 52.19. A sketch created with Silk (top left) and its automatic transformation into a Motif user interface (top right). A storyboard (bottom) used to test sequences of interactions, here a button that rotates an object. (© J. Landay, with permission)

recognition, Silk interprets the marks as widgets, annotations, and so forth. Even in its sketched form, the user interface is functional: buttons can be pressed, tools can be selected in a toolbar, and so on. The sketch can also be turned into an actual interface, i.e., using the Motif toolkit. Finally, storyboards can be created to describe and test sequences of interactions. Monet (Li & Landay, 2005) expands this approach by supporting the specification of animations and continuous interaction such as drag-and-drop. Silk and Monet therefore combine some aspects of offline and online prototyping techniques, trying to get the best of both worlds. Another example is Denim (Lin, Newman, Hong, & Landay, 2000), which addresses the prototyping of websites. Like Silk, it uses a sketch-based interface to allow designers to quickly enter ideas about the overall design of the website as well as individual pages and test the navigation. This illustrates a current trend in research where online tools attempt to support not only the development of the final system, but the whole design process. The following section will also illustrate this trend in the new and very active areas of mixed reality and pervasive computing.

PROTOTYPING MIXED REALITY AND PERVASIVE COMPUTING SYSTEMS

While most examples in the previous sections concerned traditional graphical user interfaces (GUIs), there is an increasing need to address the design and prototyping of systems that combine the physical and online worlds. The current trend towards pervasive computing and mixed reality started with Weiser's (1991) seminal article on ubiquitous computing and was carried on by augmented reality (Wellner, Mackay, & Gold, 1993) and tangible interfaces (Holmquist, Schmidt, & Ullmer, 2004). A common theme of these approaches is to combine interaction with the real world and interaction with online information, taking advantage of humans' abilities to interact with physical artifacts.

While mixed reality emphasizes the role of physical objects, pervasive computing emphasizes the role of the physical space. Both raise new and difficult issues when trying to actually design and build such systems. At the prototyping stage, they typically require a larger effort than when designing a GUI. First, the range of possible interactions is much broader and typically includes gesture, eye-gaze, or speech. These techniques are already difficult to incorporate into traditional interfaces: in a mixed reality or pervasive computing environment, the user is much less constrained, making recognition and interpretation of users' actions much harder. Designers must then address recognition errors and more generally the effects of context on the sensing techniques. Second, the range of artifacts to design is larger than with desktop interfaces, as it includes the physical artifacts that the users may manipulate; in pervasive computing, the role of user location, the issues of wireless network coverage and sensor range, the difficulty of providing feedback where needed are all new challenges that need to be addressed. Finally, these systems are not necessarily used for tasks where speed of execution and productivity are the primary measures of success. Whether they are used in the home, for tourism, such as tour guides, for assisted living or plain entertainment, the wider and less well-mapped design space typically require extensive prototyping work.

For example, Boucher and Gaver (2006) described their experiences in designing the drift table, a coffee table with a small screen in the center displaying an aerial view of England and load sensors measuring the weight of objects on the table. According to the measured weight and its estimated location, the aerial view slowly drifts in that direction, as if navigating with a hot air balloon. While the resulting design is conceptually simple, it required a many prototypes, both to explore what it should do and how it could be built. The final prototype (Fig. 52.20) had to be aesthetically pleasing as well as fully functional for testing in people's homes. This example shows that the higher precision prototypes had to address interaction, function, and looks simultaneously. This proved particularly challenging but is common when prototyping these types of systems.

The prototyping strategies that we have described, such as horizontal, vertical, and scenario-based prototypes, are still valid for prototyping mixed reality and pervasive computing systems, as well as some of the prototyping techniques, such as offline prototyping with mock ups and video prototyping. Online prototyping, however, requires specific tools and iterative and incremental prototyping, even more so. The rest of this section explores some of the existing tools.

FIGURE 52.20. The drift table (© W. Gaver, with permission)

Phidgets (Greenberg & Fitchett, 2001) are physical sensors and actuators that can be controlled from a computer through a USB connection. A wide range of sensors is available (light, motion, distance, pressure, humidity, etc.), as well as input devices (buttons, sliders), LEDs, LCD displays, motors, and RFID tag readers. By hiding their implementations and exposing their functionalities through a standard software interface, phidgets can greatly facilitate the design and prototyping of tangible interfaces. For example, they can be embedded into a physical artifact and programmed from the computer to prototype a mobile device. Since all phidgets can be represented on the computer screen, they can also be used for Wizard of Oz settings, where the Wizard can directly control the output phidgets such as LEDs and motors.

Papier-Mâché (Klemmer, Li, Lin, & Landay, 2004) is a toolkit for creating tangible interfaces based on vision, RFID tags, and barcodes. It is more adapted to the development of augmented paper applications than Phidgets and provides a richer development environment. The DART toolkit (MacIntyre, Gandy, Dow, & Bolter, 2004) takes a similar approach, but targets augmented and mixed reality applications. The development environment is based on Macromedia Director, with the explicit goal of being familiar to designers. In the context of prototyping, both toolkits can be used for applications other than those that they explicitly target. For example, DART can superimpose a computer-generated model with live or recorded video, which can be used to put an image or 3-D model of a device being designed in the real world.

Topiary (Li, Hong, & Landay, 2004) is a tool for prototyping pervasive applications that use the locations of people, places, and objects. Unlike the above tool, its goal is not to create real-world prototypes allowing to test a system in situ. Rather, it is an online tool that allows to create and test scenarios on the screen, using maps representing the environment and icons representing the people, places, and objects of interest. Once a scenario has been created, the tool allows the user to experience it either from a bird's eye view or from the perspective of a particular user by displaying the user's PDA. Such scenario-based prototyping can be very useful in the early stages of design, when exploring ideas.

Finally, a CAPpella (Dey, Hamid, Beckmann, Li, & Hsu, 2004), is a tool to help end users create or customize context-aware applications. It uses programming by demonstration so that users only have to show examples of sensor data and the system's reaction. While the system targets end users, it can also be used by designers to create prototypes of context-aware applications. By hiding the complexity of programming recognizers, it allows for a fast exploration of alternatives.

In summary, while mixed reality and pervasive computing create new challenges for designers, a number of tools are starting to appear to help with the design process in general and prototyping in particular. Interestingly, these tools can also be used to prototype more traditional applications. For example, phidgets can be used to control an online application in a Wizard of Oz setting, DART can be used to create scenarios that mix videos with models of future artifacts, and a CAPpella can be used to experiment with recognition-based applications. As mixed reality and pervasive computing systems become more widely available, they will no doubt provide more tools to be used by designers in unexpected ways to help with the prototyping of interactive systems at large.

CONCLUSION

Prototyping is an essential component of interactive system design. Prototypes may take many forms, from rough sketches to detailed working prototypes. They provide concrete representations of design ideas and give designers, users, developers and managers an early glimpse into how the new system will look and feel. Prototypes increase creativity, allow early evaluation of design ideas, help designers think through and solve design problems, and support communication within multidisciplinary design teams.

Prototypes, because they are concrete and not abstract, provide a rich medium for exploring a design space. They suggest alternate design paths and reveal important details about particular design decisions. They force designers to be creative and to articulate their design decisions. Prototypes embody design ideas and encourage designers to confront their differences of opinion. The precise aspects of a prototype offer specific design solutions: designers can then decide to generate and compare alternatives. The imprecise or incomplete aspects of a prototype highlight the areas that must be refined or require additional ideas.

We defined prototypes and then discussed them as design artifacts. We introduced four dimensions by which they can be analyzed: representation, precision, interactivity, and evolution. We then discussed the role of prototyping within the design process and explained the concept of creating, exploring, and modifying a design space. We briefly described techniques for generating new ideas to expand the design space and techniques for choosing among design alternatives to contract the design space.

We described a variety of rapid prototyping techniques for exploring ideas quickly and inexpensively in the early stages of design, including offline techniques (from paper and pencil to video) and online techniques (from fixed to interactive simulations). We then described iterative and evolutionary prototyping techniques for working out the details of the online interaction, including development tools and software environments. Finally, we addressed the emerging fields of mixed reality and pervasive computing and described the new challenges they raise for interactive systems design, as well as some of the tools available for prototyping them.

We view design as an active process of working with a design space, expanding it by generating new ideas and contracting as design choices are made. Prototypes are flexible tools that help designers envision this design space, reflect upon it, and test their design decisions. Prototypes are diverse and can fit within any part of the design process, from the earliest ideas to the final details of the design. Perhaps most important, prototypes provide one of the most effective means for designers to communicate with each other, as well as with users, developers, and managers, throughout the design process.

References

Apple Computer. (1996). *Programmer's guide to MacApp.* Cupertino, CA: Apple Computer, Inc.

Beaudouin-Lafon, M. (2000). Instrumental interaction: An interaction model for designing post-WIMP user interfaces. *Proceedings ACM Human Factors in Computing Systems, CHI'2000*, The Hague, Netherlands, 446–453.

Beaudouin-Lafon, M. (2001). Novel interaction techniques for overlapping Windows. *Proceedings of ACM Symposium on User Interface Software and Technology, UIST 2001*, Orlando, Florida. *CHI Letters*, 3(2), 153–154.

Beaudouin-Lafon, M. (2004, May). Designing interaction, not interfaces. *Proceedings of the Working Conference on Advanced Visual interfaces, AVI'04* (pp. 15–22). Gallipoli, Italy.

Beaudouin-Lafon, M., & Lassen, M. (2000). The architecture and implementation of a post-WIMP graphical application. *Proceedings of ACM Symposium on User Interface Software and Technology, UIST 2000,* San Diego, CA. *CHI Letters 2*(2), 191–190.

Beaudouin-Lafon, M., & Mackay, W. (2000, May). Reification, polymorphism and reuse: Three principles for designing visual interfaces. *Proceedings Conference on Advanced Visual Interfaces, AVI 2000*, (pp. 102–109). Palermo, Italy.

Beck, K. (2000). *Extreme programing explained*. New York: Addison-Wesley.

Bederson, B., & Hollan, J. (1994). Pad++: A zooming graphical interface for exploring alternate interface physics. *Proceedings of ACM Symposium on User Interface Software and Technology, UIST'94*, Marina del Rey, 17–26.

Bederson, B., & Meyer, J. (1998). Implementing a zooming interface: Experience building Pad++. *Software Practice and Experience, 28*(10), 1101–1135.

Bederson, B. B., Meyer, J., & Good, L. (2000). Jazz: An extensible zoomable user interface graphics toolkit in Java. *Proceedings of ACM Symposium on User Interface Software and Technology, UIST 2000,* San Diego, CA. *CHI Letters 2*(2), 171–180.

Berger, N. (2006). The excel story. *Interactions, 13*(1),14–17.

Bier, E., Stone, M., Pier, K., Buxton, W., & De Rose, T. (1993). Toolglass and magic lenses: The see-through interface. *Proceedings ACM SIGGRAPH* (pp. 73–80). Anaheim, CA.

Bødker, S., (1999). Scenarios in user-centered design: Setting the stage for reflection and action. *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, HICSS-32* (Vol 3, article 3053, 11 pages). Wailea, HI: IEEE Computer Society.

Bødker, S., Ehn, P., Knudsen, J., Kyng, M., & Madsen, K. (1988). Computer support for cooperative design. *Proceedings of the CSCW'88 ACM Conference on Computer-Supported Cooperative Work, (pp.* 377–393). Portland, OR.

Boehm, B. (1988). A spiral model of software development and enhancement. *IEEE Computer, 21*(5), 61–72.

Boucher, A., & Gaver, W. (2006). Developing the drift table. *Interactions, 13*(1), 24–27.

Chapanis, A. (1982). Man/Computer research at Johns Hopkins. In R. A. Kasschau, R. Lachman, & K. R. Laughery (Eds.), *Information technology and psychology: Prospects for the future* (238–249). New York, NY: Praeger Publishers, Third Houston Symposium.

Chapuis, O., & Roussel, N. (2005, October). Metisse is not a 3D desktop! *Proceedings of the 18th Annual ACM Symposium on User interface Software and Technology, UIST '05* (pp. 13–22). Seattle, WA.

Collaros, P. A., & Anderson, L. R. (1969). Effect of perceived expertness upon creativity of members of brainstorming groups. *Journal of Applied Psychology, 53*, 159–163.

de Vreede, G.-J., Briggs, R., van Duin, R., & Enserink, B. (2000). Athletics in electronic brainstorming: Asynchronous brainstorming in very large groups. *Proceedings of the 33nd Annual Hawaii International Conference on System Sciences, HICSS-33* (Vol 1, article 1042, 11 pages). Wailea, HI: IEEE Computer Society.

Dey, A. K., Hamid, R., Beckmann, C., Li, I., & Hsu, D. (2004, April). A CAPpella: Programming by demonstration of context-aware applications. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '04* (pp. 33–40). Vienna, Austria.

Diehl, M., & Stroebe, W. (1987). Productivity loss in brainstorming groups: Towards the solution of a riddle. *Journal of Personality and Social Psychology, 53*(3), 497–509. Washington, D.C.: American Psycology Association.

Dijkstra-Erikson, E., Mackay, W. E., & Arnowitz, J. (2001, March). Trialogue on design of. *ACM/Interactions,* 109–117.

Dourish, P. (1997). Accounting for system behaviour: Representation, reflection and resourceful action. In M. Kyng, & L. Mathiassen, (Eds.), *Computers and design in context* (pp. 145–170). Cambridge, MA: MIT Press.

Eckstein, R., Loy, M., & Wood, D. (1998). *Java swing*. Cambridge, MA: O'Reilly.

Fekete, J.-D. (2004). The InfoVis toolkit. *Proceedings of the 10th IEEE Symposium on Information Visualization, InfoVis'04, IEEE Press* (pp. 167–174) Austin, Texas.

Frishberg, N. (2006). Prototyping with junk. *Interactions, 13*(1), 21–23.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns, elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

Good, M. D., Whiteside, J. A., Wixon, D. R., & Jones, S. J. (1984). Building a user-derived interface. *Communications of the ACM, 27*(10), 1032–1043. New York: ACM.

Goodman, D. (1987). *The complete HyperCard handbook*. New York, NY: Bantam Books.

Greenbaum, J., & Kyng, M. (Eds.). (1991). *Design at work: Cooperative design of computer systems*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Greenberg, S., & Fitchett, C. (2001, November). Phidgets: Easy development of physical interfaces through physical widgets. *Proceedings of the 14th Annual ACM Symposium on User interface Software and Technology, UIST '01* (pp. 209–218). Orlando, FL.

Holmquist, L. E., Schmidt, A., & Ullmer, B. (2004). Tangible interfaces in perspective. *Personal and Ubiquitous Computing, 8*(5), 291–293. Heidelberg: Springer.

Hong, J. I., & Landay, J. A. (2000, November). SATIN: A toolkit for informal ink-based applications. *Proceedings of the 13th Annual ACM Symposium on User interface Software and Technology, UIST '00* (pp. 63–72). San Diego, CA.

Houde, S., & Hill, C. (1997). What do prototypes prototype? In M. G. Helaner, T. K. Landauer. & P. V. Pradhu (Eds.) *Handbook of human computer interaction* (2nd ed.) (pp. 367–381). North-Holland

Hudson, S. E., Mankoff, J., & Smith, I. (2005, April). Extensible input handling in the subArctic toolkit. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '05* (pp. 381–390). Portland, Oregon.

Kelley, J. F. (1983). An empirical methodology for writing user-friendly natural language computer applications. *Proceedings of CHI '83 Conference on Human Factors in Computing Systems.* Boston, MA, 193–196.

Klemmer, S. R., Li, J., Lin, J., & Landay, J. A. (2004, April). Papier-mache: Toolkit support for tangible input. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '04* (pp. 399–406). Vienne, Austria.

Krasner, E. G., & Pope, S. T. (1988, August/September). A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 27–49.

Kurtenbach, G., Fitzmaurice, G., Baudel, T., & Buxton. W. (1993). The design of a GUI paradigm based on tablets, two-hands, and transparency. *Proceedings of ACM Human Factors in Computing Systems, CHI'97* (pp. 35–42). Atlanta, GA.

Landay, J., & Myers, B. A. (2001). Sketching interfaces: Toward more human interface design. *IEEE Computer, 34*(3), 56–64.

Li, Y., & Landay, J. A. (2005, October). Informal prototyping of continuous graphical interactions by demonstration. *Proceedings of the 18th Annual ACM Symposium on User interface Software and Technology, UIST '05* (pp. 221–230). Seattle, WA.

Li, Y., Hong, J. I., & Landay, J. A. (2004, October). Topiary: A tool for prototyping location-enhanced applications. *Proceedings of the 17th Annual ACM Symposium on User interface Software and Technology, UIST '04* (pp. 217–226). Santa Fe, NM.

Lin, J., Newman, M. W., Hong, J. I., & Landay, J. A. (2000, April). DENIM: Finding a tighter fit between tools and practice for website design. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '00* (pp. 510–517). Hague, Netherlands.

Linton, M. A., Vlissides, J. M., & Calder, P. R. (1989). Composing user interfaces with InterViews. *IEEE Computer, 22*(2), 8–22.

MacIntyre, B., Gandy, M., Dow, S., & Bolter, J. D. (2005). DART: A toolkit for rapid design exploration of augmented reality experiences. *ACM Transactions on Graphics, 24*(3), 932–932.

Mackay, W.E. (1988). Video prototyping: A technique for developing hypermedia systems. *Conference Companion of ACM CHI'88, Conference on Human Factors in Computing. Washington DC.* Retrieved April 2, 2007, from http://www.lri.fr/~mackay/publications.html.

Mackay, W. E. (1995). Ethics, lies and videotape. *Proceedings ACM Human Factors in Computing Systems, CHI'95* (pp. 138–145). Denver, CO.

Mackay, W. E. (2000). Video techniques for participatory design: Observation, brainstorming & prototyping. *Tutorial Notes, CHI 2000, Human Factors in Computing Systems,* Hague, Netherlands. Retrieved April 2, 2007 from http://www.lri.fr/~mackay/publications.html.

Mackay, W.E. (2002). Using video to support interaction design, DVD tutorial, INRIA and ACM/SIGCHI. Retrieved April 2, 2007 from http://stream.cc.gt.atl.ga.us/hccvideos/viddesign.php

Mackay, W. E., & Fayard, A.-L. (1997). HCI, natural science and design: A framework for triangulation across disciplines. *Proceedings of ACM DIS '97, Designing Interactive Systems* (pp. 223–234). Pays-Bas, Amsterdam.

Mackay, W. E., & Pagani, D. (1994). Video mosaic: Laying out time in a physical space. *Proceedings of ACM Multimedia '94* (pp. 165–172). San Francisco, CA.

Mackay, W., Fayard, A. L., Frobert, L., & Médini, L. (1998) Reinventing the familiar: Exploring an augmented reality design space for air traffic Control. *Proceedings of ACM CHI '98 Human Factors in computing Systems* (pp. 558–565). Los Angeles, CA.

Mackay, W., Ratzer, A., & Janecek, P. (2000). Video artifacts for design: Bridging the gap between abstraction and detail. *Proceedings ACM Conference on Designing Interactive Systems, DIS 2000* (pp. 72–82). New York, NY.

Muller, M. J. (1991). PICTIVE: An exploration in participatory design. *Proceedigs of ACM CHI'91 Human Factors in Computing Systems* (pp. 225–231). New Orleans, LA.

Myers, B. A., & Rosson, M. B. (1992). Survey on user interface programming. *ACM Conference on Human Factors in Computing Systems, CHI'92* (pp. 195–202). Monterey, CA.

Myers, B. A., Giuse, D. A., Dannenberg, R. B., Vander Zander, B., Kosbie, D. S., Pervin, E., et al. (1990). Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer, 23*(11), 71–85.

Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., et al. (1997). The Amulet environment. *IEEE Transactions on Software Engineering, 23*(6), 347–365.

NeXT Corporation. (1991). *NeXT interface builder reference manual*. Redwood City, CA: NeXT Corporation.

Norman, D. A., & Draper S. W. (Eds.). (1986). *User centered system design*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Osborn, A. (1957). *Applied imagination: Principles and procedures of creative thinking* (Rev. ed.). New York, NY: Scribner's.

Ousterhout, J. K. (1994). *Tcl and the Tk toolkit*. Reading, MA: Addison Wesley.

Perkins, R., Keller, D. S., & Ludolph, F (1997). Inventing the Lisa user interface. *ACM Interactions, 4*(1), 40–53.

Rettig, M. (1994). Prototyping for tiny fingers. *Communication ACM, 37*(4), 21–27.

Raskin, J. (2000). *The humane interface*. New York, NY: Addison-Wesley.

Roseman, M., & Greenberg, S. (1996). Building real-time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer-Human Interaction, 3*(1), 66–106.

Roseman, M., & Greenberg, S. (1999). Groupware toolkits for synchronous work. In M. Beaudouin-Lafon (Ed.), *Computer-supported cooperative work*: *Trends in software series* (pp. 135–168). Chichester: Wiley.

Schroeder, W., Martin, K., & Lorensen, B. (1997). *The visualization toolkit*. Upper Saddle River, NJ: Prentice Hall.

Snyder, C. (2003). *Paper prototyping: The fast and easy way to design and refine user interfaces*. San Francisco, CA: Morgan Kaufmann.

Strass, P. (1993). IRIS inventor, a 3D graphics toolkit. *Proceedings ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, *OOPSLA'93* (pp. 192–200). Washington, DC.

Vlissides, J. M., & Linton, M. A. (1990). Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems, 8*(3), 237–268.

Weiser, M. (1991) The computer for the 21st century, *Scientific American, 265*(3), 94–104.

Wellner, P., Mackay, W. E. & Gold, R., (Eds.), (July 1993). Special Issue on Computer-Augmented Environments. *Communications of the ACM*. New York: ACM.