Chapter 52

Prototyping Tools and Techniques

Michel Beaudouin-Lafon, Université Paris-Sud, mbl@lri.fr

Wendy E. Mackay, INRIA, wendy.mackay@inria.fr

1. Introduction

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

Design is about making choices. In many fields that require creativity and engineering skill, such as architecture or 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 how it is expanded and contracted by generating and selecting design ideas. The next three sections describe specific prototyping approaches: Rapid prototyping, both off-line and on-line, for early stages of design, iterative prototyping, which uses on-line development tools, and evolutionary prototyping, which must be based on a sound software architecture.

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 reflect upon the final system.

Note that prototypes may be defined differently in other fields. For example, an architectural prototype is a scaled-down model of the final building. This is not possible for interactive system prototypes: the designer may limit the amount of information the prototype can handle, but the actual interface must be presented at full scale. Thus, a prototype interface to a database may handle only a small pseudo database but must still present a full-size display and interaction techniques. Full-scale, one-of-a-kind models, such as a hand-made dress sample, are another type of prototype. These usually require an additional design phase in order to mass-produce the final design. Some interactive system prototypes begin as one-of-a-kind models which 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 pre-defined 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.

Human-Computer Interaction is a multi-disciplinary field which combines elements of science, engineering and design (Mackay and Fayard, 1997, Djkstra-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.

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

2.1 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: off-line and on-line.

Off-line prototypes (also called *paper prototypes*) do not require a computer. They include paper sketches, illustrated story-boards, cardboard mock-ups and videos. The most salient characteristics of off-line 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 purpose.

On-line 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 on-line prototypes is usually higher, and may require skilled programmers to implement advanced interaction and/or 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 already 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 twenty years of prototyping, in both research and industrial settings, we have yet to find a situation in which this is true.

First, off-line prototypes are very inexpensive and quick. This permits a very rapid iteration cycle and helps prevent the designer from becoming overly attached to the first possible solution. Off-line prototypes make it easier to *explore the design space* (see section 3.1), examining a variety of design alternatives and choosing the most effective solution. On-line prototypes introduce an intermediary between the idea and the implementation, slowing down the design cycle.

Second, off-line 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 creative design.

Finally and perhaps most importantly, off-line 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 story-board or cardboard mock-up requires no particular 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 off-line prototypes, they are not a panacea. In some situations, they are insufficient to fully evaluate 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, off-line 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 and Hill, 1997) at each stage of the design process and choose the representation that is best suited to the current design question.

2.2 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. 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 wigglely shape of an object or a squiggle that represents a label are directly perceived as imprecise. This is more difficult to achieve with an on-line 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.

2.3 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 web sites) 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 about a decade. Another problem is that the quality of interaction is tightly linked to the end users and a deep understanding of their work practices: 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.

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 on-line prototypes, in fact it is often easier to explore different interaction strategies with off-line 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. Or, one may create a very precise but non-interactive

[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.

prototype, such as a detailed animation that shows feedback from a specific action by a user.

Prototypes can support interaction in various ways. For off-line 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 on-line 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 section 4.1.) 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 pre-computed animations, are non-interactive: the user cannot interact, or pretend to interact, with it. Fixed prototypes are often used to illustrate or test scenarios (see chapter 53). *Fixed-path prototypes* support limited interaction. The extreme case is a fixed prototype in which each step is triggered by a pre-specified 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, she presents the corresponding screen showing a dialog box. When the user points to the word "OK", 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 in the role of the user can get a feel for 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 also be used for horizontal and task-based prototypes (see section 3.1).

Open prototypes support large sets of interactions. Such prototypes work like the real system, with some limitations. They usually only cover part of the system (see vertical prototypes, section 3.1), 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 be like, but only in pre-specified situations. Open prototypes allow designers to test a wide range of examples of how users will interact with the system.

2.4 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 off-line or on-line. Creating precise software prototypes, even if they must be re-implemented in the final version of the system, is important for detecting and fixing interaction problems. Section 4 presents specific prototyping techniques, both off-line and on-line.

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. Section 5 describes tools and techniques for creating iterative prototypes.

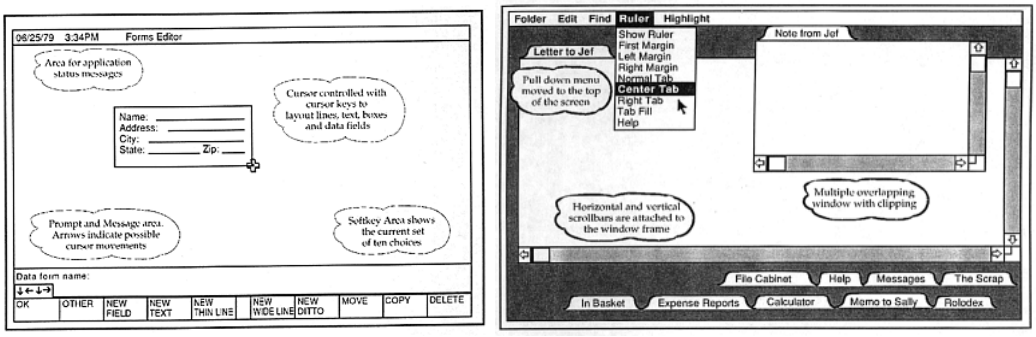

Figure 1: Evolutionary prototypes of the Apple Lisa:
July 1979 (left), October 1980 (right) (Perkins et al., 1997) [[need permission]]

Evolutionary prototypes are a special case of iterative prototypes in which the prototype evolves into part or all of the final system (Fig.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. Section 6 describes how to create evolutionary prototypes, by building upon software architectures specifically designed to support interactive systems.

3. 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, i.e. 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 Human-Computer Interaction 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 (see chapters 48-50 in this volume) to testing and evaluation (see chapters 56-59 in this volume). Prototypes support this goal by allowing users see and experience the final system long before it is built. Designers can identify functional requirements, 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 (re)analyze 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 (also called Cooperative) Design is a form of user-centered design that actively involves the user in all phases the design process (see Greenbaum and Kyng, 1991, and chapter 54 in this volume.) 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. 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.

3.1 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. Designers do, of course, benefit from scientific research findings and they may use scientific methods to evaluate interactive systems. But designers also require specific techniques for generating new ideas and balancing complex sets of trade-offs, to help them develop and refine design ideas.

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

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 process is iterative: more cyclic, than reductionist. That is, the designer does not begin with a rough idea and successively add more precise details until the final solution is reached. Instead, she begins with a design problem, which imposes set of constraints, and generates a set of ideas to form the initial design space. 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 and 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.

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, 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 and Anderson, 1969, Diehl and 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 purposes, 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) point out, one should also consider elaboration of ideas, as group members react to each other's ideas.

We have found that brainstorming, including a variety of variants, is an important group-building exercise and participatory design. Designers may, of course, brainstorm ideas by themselves. But 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 pre-specified 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 pre-specified period of time. 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 that involves prototypes called video brainstorming (Mackay, 2000): participants not only write or draw their ideas, they act them out in front of a video camera (Fig. 2). The goal is the same as other brainstorming exercises, i.e. to create as many new ideas as possible, without critiquing them. The use of video, combined with paper or cardboard mock-ups, encourages participants to actively experience the details of the interaction and to understand each idea from the perspective of the user.

Each video brainstorming idea takes 2-5 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 context in which the ideas were created.)

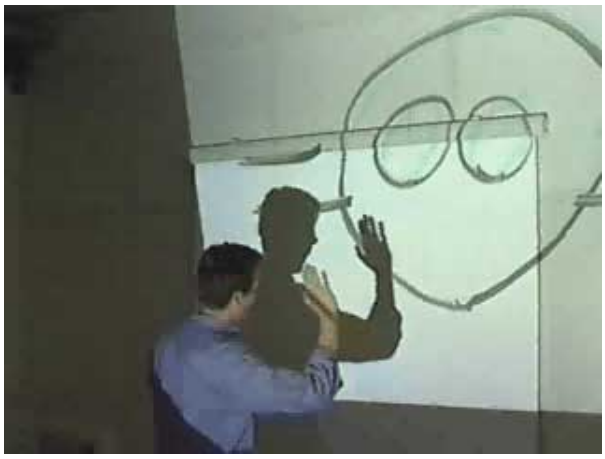

Figure 2: Video Brainstorming: One person moves the transparency, projected onto the wall, in response to the actions of the user, who explores how he might interact with an on-line animated character. Each interaction idea is recorded and videotaped.

Video brainstorming requires thinking more deeply about each idea. It is easier to stay abstract 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 seriously consider how a user would interact with the idea. It also encourages designers and users to think about new ideas in the context in which they will be used. Video clips from a video brainstorming session, even though rough, are much easier for the design team, including developers, to interpret than ideas from a standard brainstorming session.

We generally run a standard brainstorming session, either oral or with cards, prior to a video brainstorming session, 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 on 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 can 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 (see chapter 56) 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 take other factors into account 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 (see chapters 1-6). They can also evaluate their designs with respect to ergonomic criteria (see chapter 51) or design principles (Beaudouin-Lafon and Mackay, 2000). See chapters 56-60 for a more thorough discussion of testing and evaluation methods.

Another strategy is to create one or more scenarios (see chapter 53) 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 of time. Good scenarios involve more than a string of independent tasks; they should incorporate real-world activities, including common or repeated tasks, successful activities and break-downs 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 (off-line or on-line) described in section 4.1. They 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. See section 4.1 for more information on how to develop scenarios, storyboard and then videotape them.

To an outsider, 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.

3.2 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 sections 4, 5 and 6.

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 (see section 5.1), 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. As a consequence, 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 and Mackay, 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 the we could implement a sufficiently fast response.

Vertical prototypes are generally high precision, software prototypes because their goal is to validate an idea 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 (see chapter 48), 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 off-line and on-line 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 (see chapter 53). A good scenario includes both common and unusual situations, and should explore patterns of activity over time. Bødker (1995) has developed a checklist to ensure that no important issues have been left out.

We find it useful to begin with *use scenarios* based on observations of or interviews with real users. Ideally, some of those users should participate in the creation of the specific scenarios, 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 new 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 situation, even if it addresses only a subset of the planned functionality.

Section 4 describes a variety of rapid prototyping techniques which can be used in any of these four prototyping strategies. We begin with off-line rapid prototyping techniques, followed by on-line prototyping techniques.

4. Rapid prototypes

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.

How rapid is rapid depends on the context of the particular project and the stage in the design process. Early prototypes, e.g. 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 introduced in section 2. Off-line techniques are generally more rapid than on-line one. However, creating successive iterations of an on-line prototype may end up being faster than creating new off-line prototypes.

4.1 Off-line rapid prototyping techniques

Off-line 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 & pencil

The fastest form of prototyping involves paper, transparencies and post-it notes to represent aspects of an interactive system (for an example, see 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.

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 pre-printed, overlaid on each other as necessary) to create an interactive display on the wall. The user can interact by pointing (Fig. 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.

Figure 3: Hand-drawn transparencies can be projected onto a wall, creating an interface a user can respond to.

Note that most paper prototypes begin with quick sketches on paper, then progress to more carefully-drawn screen images made with a computer (Fig. 4). In the early stages, the goal is to generate a wide range of ideas and expand the design space, not determine the final solution. Paper and pencil prototypes are an excellent starting point for horizontal, task-based and scenario-based prototyping strategies.

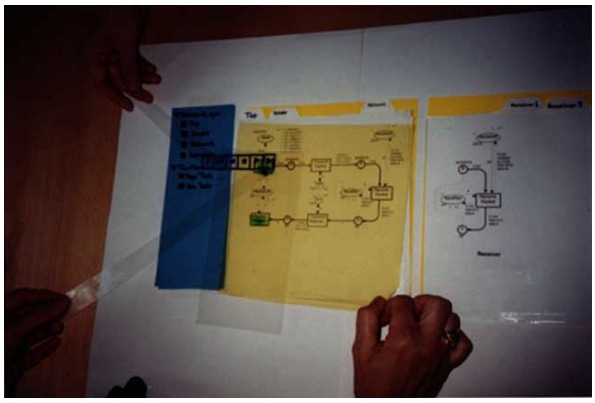

Figure 4: Several people work together to simulate interacting with this paper prototype. One person moves a transparency with a mouse pointer while another moves the diagram accordingly.

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 et al., 1988). Generally made of cardboard, foamcore or other found materials, mock-ups are physical prototypes of the new system. Fig. 5 shows an example of a hand-held mockup showing the interface to a new hand-held device. The mock-up provides a deeper understanding of how the interaction will work in real-world situations than possible with sets of screen images.

Figure 5: Mock-up of a hand-held display with carrying handle.

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 vs. 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. 6).

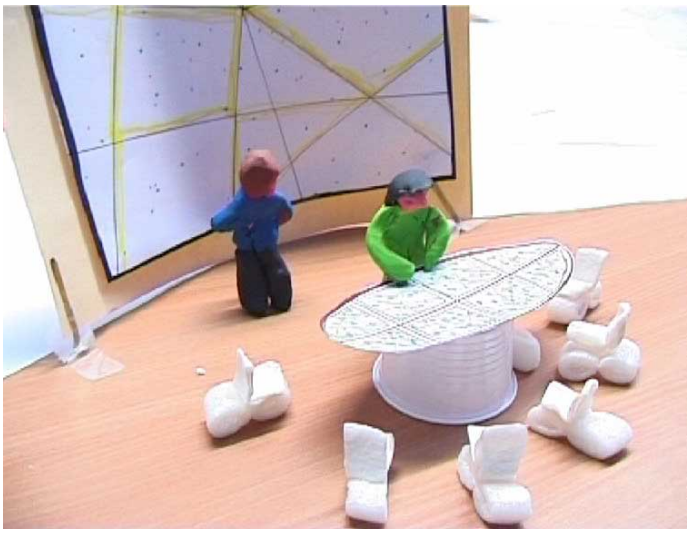

Figure 6: Scaled mock-up of an air traffic control table, connected to a wall display.

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 (1993) dubbed this technique the *Wizard of Oz* , 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 terminal and interacts with a program. Hidden elsewhere, the software designer (the wizard) watches what the user does and, by responding in different ways, 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 was initially used to develop natural language interfaces (e.g. Chapanis, 1982, Wixon, Whiteside, Good and Jones, 1993). 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 section 3.1, they differ from video brainstorming in that the goal is to refine a single design, not generate new ideas.

Video prototypes may build on paper & pencil prototypes and cardboard mock-ups and can also 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 dialog (or subtitles), with related annotations that explain what is happening in the scene or the type of shot (Fig. 7). A paragraph of text in a scenario corresponds to about a page of a storyboard.

<Insert image in file: figure.7.eps >

Figure 7: Storyboard. This storyboard is based on observations of real Coloured Petri Net users in a small company and illustrates how the CPN developer modifies a particular element of a net, the "Simple Protocol".

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 creating the production. Storyboards may be informal "sketches" of ideas, with only partial information. Others follow a pre-defined 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.

Storyboards can be used like comic books to communicate with other members of the design team. Designers and users can discuss the proposed system and alternative ideas for interacting with it (figure 8). Simple videos of each successive frame, with a voice over to explain what happens, can also be effective. However, 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 and Pagani, 1994) helps the designer experiment with different linear sequences and insert or delete video clips. However, the process of creating a video prototype, based on the storyboard, provides an even deeper understanding of the design.

Figure 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.

The storyboard guides the shooting of the video. We often use a technique called "editing-in-the-camera" (see Mackay, 2000) which allows us the create the video

directly, without editing later. We use title cards, as in a silent movie, to separate the clips and to make it easier to shoot. A narrator explains each event and several people may be necessary to illustrate the interaction. Team members enjoy playing with special effects, such as "time-lapse photography". For example, we can record a user pressing a button, stop the camera, add a new dialog box, and then restart the camera, to create the illusion of immediate system feedback.

Video is not simply a way to capture events in the real world or to capture design ideas, but can be a tool for sketching and visualizing interactions. We use a second live video camera as a Wizard-of-Oz tool. 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 semi-automated information. The user interacts with the objects on the screen as 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 or monitor, to create either a video brainstorm (for a quick idea) or a fully-storyboarded video prototype).

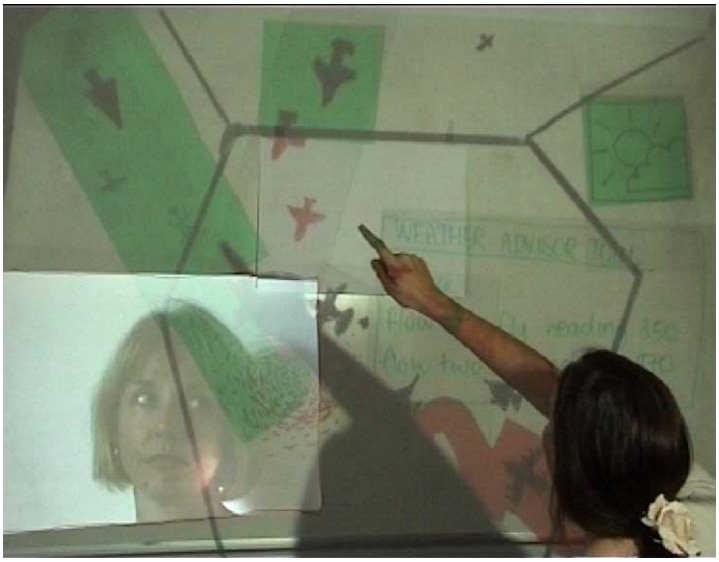

Figure 9: Complex wizard-of-oz simulation, with projected image from a live video camera and transparencies projected from an overhead projector.

Fig. 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. 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. The entire interaction is videotaped by a second video camera.

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. 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.

4.2 On-line rapid prototyping techniques

The goal of on-line rapid prototyping is to create higher-precision prototypes than can be achieved with off-line 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 off-line, such as when very dynamic interactions or visualizations are needed.

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

Non-interactive simulations

A non-interactive 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. Non-interactive simulations are usually created when off-line 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's 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 non-interactive 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; experienced developer can create sophisticated interactive simulations. However, non-interactive simulations are much faster to create. Other similar tools exist on the market, including Abvent Katabounga, Adobe AfterEffects and Macromedia Flash (Fig. 10).

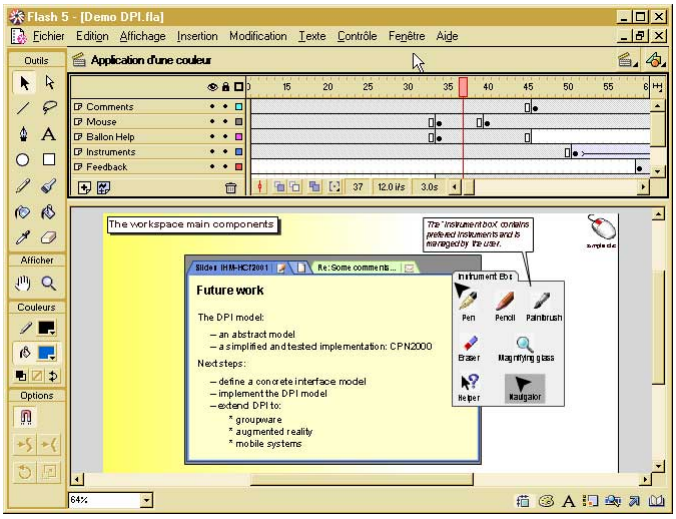

Figure 10: A non-interactive simulation of a desktop interface created with Macromedia Flash. The time-line (top) displays the active sprites while the main window (bottom) shows the animation. (O. Beaudoux, with permission)

Figure 11 shows a set of animation movies created by Dave Curbow to explore the notion of accountability in computer systems (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, etc. 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.

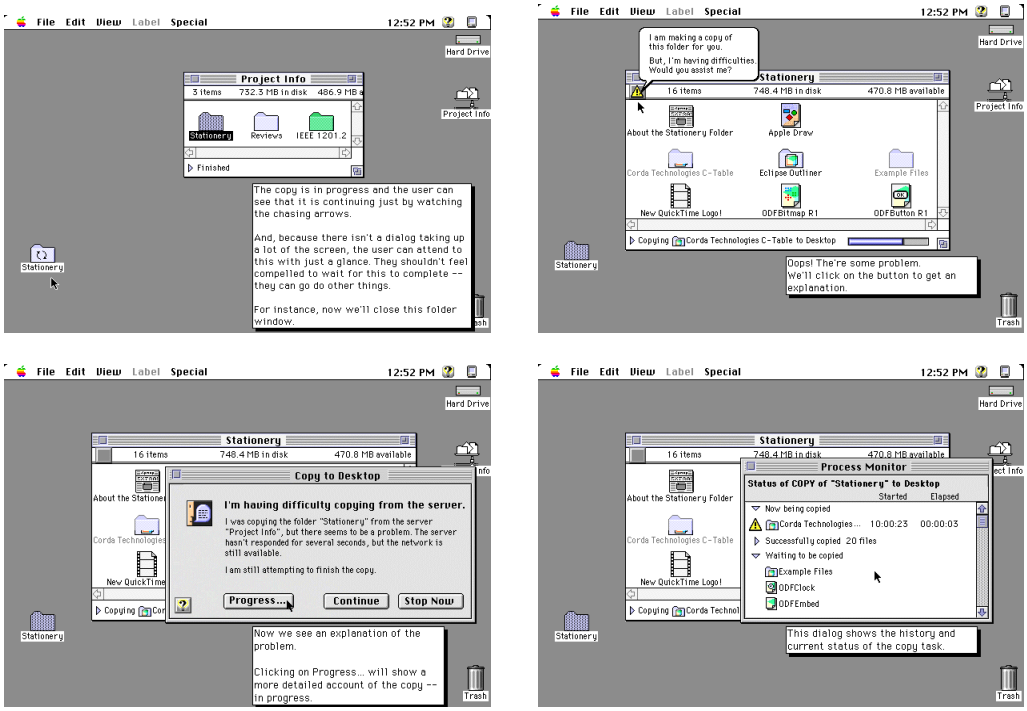

Figure 11: Frames from an animated simulation created with Macromind Director (D. Curbow, with permission)

Non-interactive animations can be created with any tool that generates images. For example, many Web designers use Adobe Photoshop to create simulations of their web sites. 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 et al., 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.

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 that the user can directly interact with. Hypercard (Goodman, 1987) is one of the most successful early prototyping tools. It is 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. 12). Stacks, cards, fields and buttons react to user events, e.g. clicking a button, as well as system events, e.g. when a new card is displayed or about to disappear (Fig. 13). Hypercard reacts according to events programmed with a scripting language called Hypertalk. For example, the following script is assigned to a button, which switches to the next card in the stack whenever the button is clicked. If this button is included in the stack background, the user will be able to browse through the entire stack:

```
on click
 goto next card
end click
```

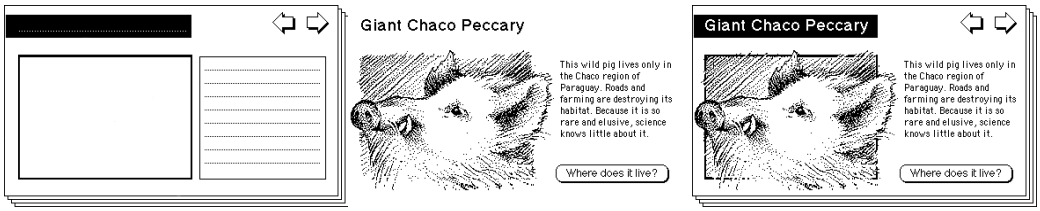

Figure 12: A Hypercard card (right) is the combination of a background (left) and the card's content (middle). *(Apple Computer, permission requested)*

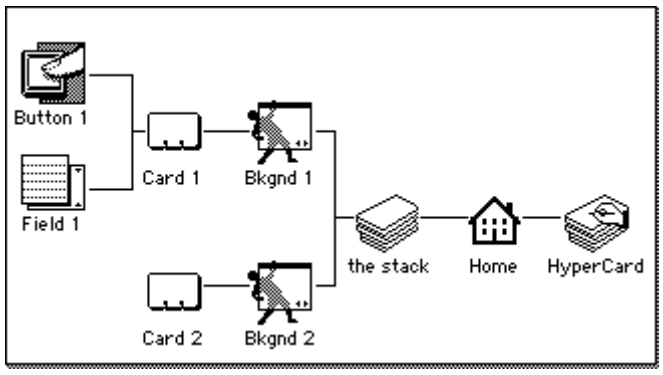

Figure 13: The hierarchy of objects in Hypercard determines the order (from left to right) in which a handler is looked up for an event *(Apple Computer, permission requested)*

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. More open interactions require more advanced use of the scripting language, but are fairly easy to master with a little practice.

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 dialog box opens. The same technique can be used to make the OK and Cancel buttons of the dialog 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.

Many educational and cultural CD-ROMs are created exclusively with Director. They often feature original visual displays and interaction techniques that would be almost impossible to create with the traditional user interface development tools described in section 5. Designers should consider tools like Hypercard and Director as user interface builders or user interface development environments. In some situations, they can even be used for evolutionary prototypes (see section 6).

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 throw-away quickly (a few hours to a few days), which may or may not be used in the final system, for performance or other technical reasons.

A scripting language is a programming language that is both lightweight and easy to learn. Most scripting languages are interpreted or semi-compiled, i.e. 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 non fatal 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 pre-existing applications or software components. Tcl (Ousterhout, 1993) was inspired by the syntax of the Unix shell, it 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.

Tcl is particularly suitable to develop user interface prototypes (or small to medium-size applications) because of its Tk user interface toolkit. Tk features all the traditional interactive objects (called “widgets”) of a UI toolkit: buttons, menus, scrollbars, lists, dialog boxes, etc. A widget is typically only one line. For example:

```
button .dialogbox.ok -text OK -command {destroy .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 5-20 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 through the use of *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. 14). 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.

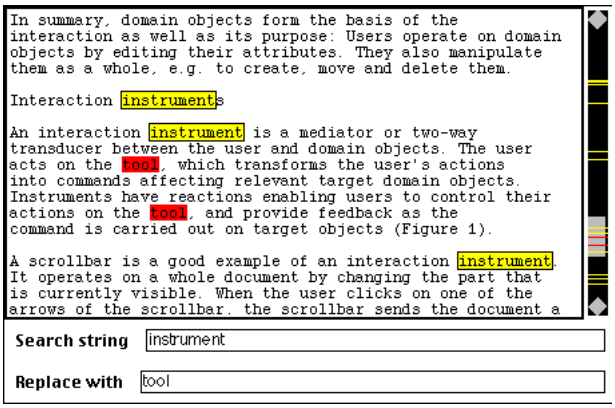

Figure 14: using the Tk text and canvas widgets to prototype a novel search and replace interaction technique (Beaudouin-Lafon, 2000).

The Tk canvas widget is a drawing surface that can contain arbitrary objects: lines, rectangles, ovals, polygons, text strings, and widgets. Tags allow behaviors (i.e. 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. 15 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 (for example, 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.

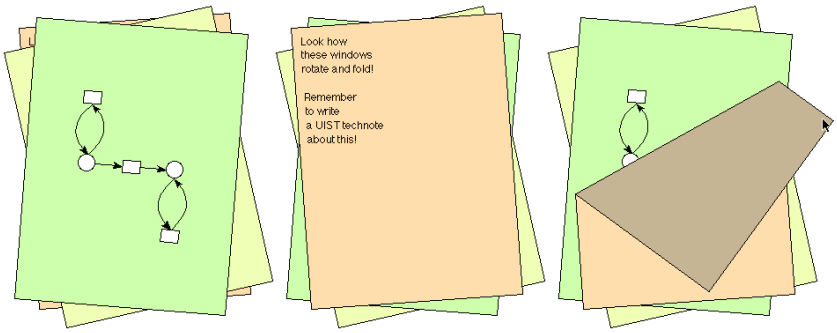

Figure 15: using the Tk canvas widget to prototype a novel window manager (Beaudouin-Lafon, 2001).

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 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 are re-implemented 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.

Software prototypes can also be used in conjunction with hardware prototypes. Figure 16 shows an example of a hardware prototype that captures hand-written 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 Tk/TCL, 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.

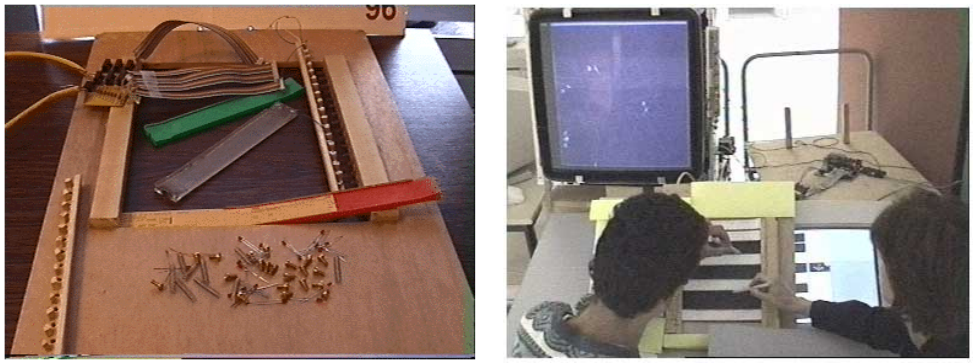

Fig. 16: Caméléon's augmented stripboard (left) is a working hardware prototype that identifies and captures hand-writing from paper flight strips. Members of the design team test the system (right), which combines both hardware and software prototypes into a single interactive simulation.

5. Iterative 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 on-line prototyping techniques described above. 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 twenty 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% - 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.

Interactive systems are inherently more powerful than non interactive ones (see Wegner, 1997, for a theoretical argument). They do not match the traditional, purely algorithmic, type of programming: an interactive system must handle user input and generate output at almost any time, whereas an algorithmic system reads input at the beginning, processes it, and displays results at the end. In addition, interactive systems must process input and output at rates that are compatible with the human perception-action loop, i.e. in time frames of 20ms to 200ms. In practice, interactive systems are both reactive and real-time systems, two active areas in computer science research.

The need to develop interactive systems more efficiently has led to two inter-related streams of work. The first involves creation of software tools, from low-level user-interface libraries and toolkits to high-level user interface development environments (UIDE). The second addresses software architectures for interactive systems: how system functions are mapped onto software modules. The rest of this section presents the most salient contributions of these two streams of work.

5.1 Software tools

Since the advent of graphical user interfaces in the eighties, a large number of tools have been developed to help with the creation of interactive software, most aimed at visual interfaces. This section presents a collection of tools, from low-level, i.e. requiring a lot of programming, to high-level.

The lowest-level tools are *graphical libraries* ,that 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* semi-automatically derive an interface from a specification of the domain objects and functions to be supported. Finally, *user interface development environments* or UIDEs provide an integrated collection of tools for the development of interactive software.

Before we describe each 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 throw-away 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 et al., 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, etc. 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 quickly change the details 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.

Graphical libraries and Window systems

Graphical libraries underlie all the other tools presented in this section. Their main purpose is to provide the developer with a hardware-independent, and sometimes cross-platform application programming interface (API) for drawing on the screen. They can be separated into two categories: direct drawing and scene-graph based. Direct drawing libraries provide functions to draw shapes on the screen, once specified their geometry and their graphical attributes. This means that every time something is to be changed on the display, the programmer has to either redraw the whole screen or figure out exactly which parts have changed. Xlib on Unix systems, Quickdraw on MacOS, Win32 GDI on Windows and OpenGL (Woo et al., 1997) on all three platforms are all direct drawing libraries. They offer the best compromise between performance and flexibility, but are difficult to program.

Scene-graph based libraries explicitly represent the contents of the display by a structure called a scene graph. It can be a simple list (called display list), a tree (as used by many user interface toolkits – see next subsection), or a direct acyclic graph (DAG). Rather than painting on the screen the developer creates and updates the scene graph, and the library is responsible for updating the screen to reflect the scene graph. Scene graphs are mostly used for 3D graphics, e.g., OpenInventor (Strass, 1993), but in recent years they have been used for 2D as well (Bederson et al., 2000, Beaudouin-Lafon & Lassen, 2000). With the advent of hardware-accelerated graphics card, scene-graph based graphics libraries can offer outstanding performance while easing the task of the developer.

Window systems provide an abstraction to allow multiple client applications to share the same screen. Applications create windows and draw into them. From the application perspective, windows are independent and behave as separate screens. All graphical libraries include or interface with a window system. Window systems also offer a user interface to manipulate windows (move, resize, close, change stacking order, etc.), called the window manager. The window manager may be a separate application (as in X-Windows), or it may be built into the window system (as in Windows), or it may be controlled of each application (as in MacOS). Each solution offers a different trade-off between flexibility and programming cost.

Graphical libraries include or are complemented by an input subsystem. The input subsystem is event driven: each time the user interacts with an input device, an event recording the interaction is added to an input event queue. The input subsystem API lets the programmer query the input queue and remove events from it. This technique is much more flexible than polling the input devices repeatedly or waiting until an input device is activated. In order to ensure that input event are handled in a timely fashion, the application has to execute an event loop that retrieves the first event in the queue and handles it as fast as possible. Every time an event sits in the queue, there is a delay between the user action and the system reaction. As a consequence, the event loop sits at the heart of almost every interactive system.

Window systems complement the input subsystem by routing events to the appropriate client application based on its focus. The focus may be specified explicitly for a device (e.g. the keyboard) or implicitly through the cursor position

(the event goes to the window under the cursor). Scene-graph based libraries usually provide a picking service to identify which objects in the scene graph are under or in the vicinity of the cursor.

Although graphical libraries and window systems are fairly low-level, they must often be used when prototyping novel interaction and/or visualization techniques. Usually, these prototypes are developed when performance is key to the success of a design. For example, a zoomable interface that cannot provide continuous zooming at interactive frame rates is unlikely to be usable. The goal of the prototype is then to measure performance in order to validate the feasibility of the design.

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: a presentation, a behavior and an application interface.

The *presentation* defines the graphical aspect of the widget. Usually, the presentation can be controlled by the application, but also externally. For example, under X-Windows, it is possible to change the appearance of widgets in any application by editing a text file specifying the colors, sizes and labels of buttons, menu entries, etc. 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: a composite widgets contains other widgets and controls their arrangement. For example menu widgets in a menu bar are stacked horizontally, while command widgets in a menu are stacked vertically. Widgets in a dialog box are laid out at fixed positions, or relative to each other so that the layout may be recomputed when the window is resized. Such constraint-based layout saves time because the interface does not need to be re-laid out completely when a widget is added or when its size changes as a result of, e.g., changing its label.

The *behavior* of a widget defines the interaction methods it supports: a button can be pressed, a scrollbar can be scrolled, a text field can be edited. The behavior also includes the various possible states of a widget. For example, most widgets can be active or inactive, some can be highlighted, etc. The behavior of a widget is usually hardwired and defines its class (menu, button, list, etc.). However it is sometimes parameterized, e.g. a list widget may be set to support single or multiple selection.

One limitation of widgets is that their behavior is limited to the widget itself. Interaction techniques that involve multiple widgets, such as drag-and-drop, cannot be supported by the widgets' behavior alone and require a separate support in the UI toolkit. Some interaction techniques, such as toolglasses or magic lenses (Bier et al., 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, e.g. with a graphical library, and is rarely justified for prototyping. Many toolkits provide a "blank" widget (Canvas in Tk, Drawing Area in Motif, JFrame in Java Swing) that 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.

The *application interface* of a widget defines how it communicate the results of the user interactions to the rest of the application. Three main techniques exist. The first and most common one is called callback functions or *callback* for short: when the widget is created, the application registers the name of a one or more functions with it. When the widget is activated by the user, it calls the registered functions (Fig. 17). The problem with this approach is that the logic of the application is split among many callback functions (Myers, 1991).

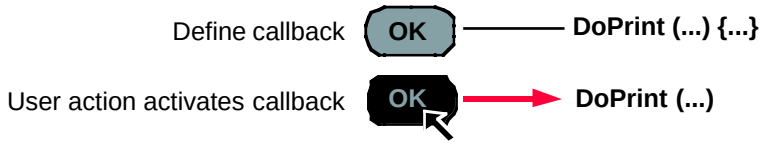

Fig. 17: Callback functions

The second approach is called *active variables* and consists of associating a widget with a variable of the application program (Fig. 18). A controller ensures that when the widget state changes, the variable is updated with a new value and, conversely, when the value of the variable changes, the widget state reflects the new value. This allows the application to change the state of the interface without accessing the widgets directly, therefore decoupling the functional core from the presentation. In addition, the same active variable can be used with multiple widgets, providing an easy way to support multiple views. Finally, it is easier to change the mapping between widgets and active variables than it is to change the assignment of callbacks. This is because active variables are more declarative and callbacks more procedural. Active variables work only for widgets that represent data, e.g. a list or a text field, but not for buttons or menus. Therefore they complement, rather than replace, callbacks. Few user interface toolkits implement active variables. Tcl/Tk (Ousterhout, 1994) is a notable exception.

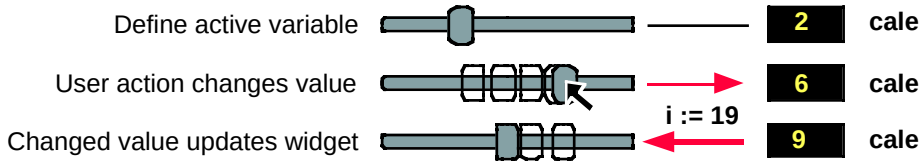

Fig. 18: Active variables

The third approach for the application interface is based on *listeners*. Rather than registering a callback function with the widget, the application registers a listener object (Fig. 19). When the widget is activated, it sends a message to its listener describing the change in state. Typically, the listener of a widget would be its model (using the MVC terminology) or Abstraction (using the PAC terminology). The first advantage of this approach therefore is to better match the most common architecture models. It is also more truthful to the object-oriented approach that underlies most user interface toolkits. The second advantage is that it reduces the "spaghetti of callbacks" described above: by attaching a single listener to several widgets, the code is more centralized. A number of recent toolkits are based on the listener model, including Java Swing (Eckstein et al., 1998).

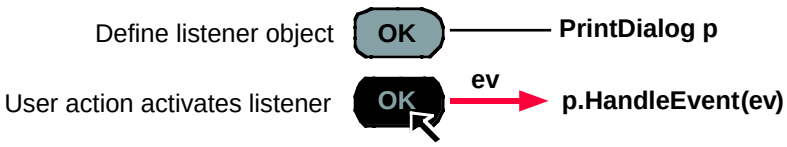

Fig. 19: Listener objects

User interface toolkits have been an active area of research over the past 15 years. InterViews (Linton et al., 1989) has inspired many modern toolkits and user interface builders. A number of toolkits have also been developed for specific applications such as groupware (Roseman and Greenberg, 1996, 1999) or visualization (Schroeder et al., 1997).

Creating an application or a prototype with a UI toolkit requires a solid knowledge of the toolkit and experience with programming interactive applications. In order to control the complexity of the inter-relations between independent pieces of code (creation of widgets, callbacks, global variables, etc.), it is important to use well-known design patterns. Otherwise the code quickly becomes unmanageable and, in the case of a prototype, unsuitable to design space exploration. Two categories of tools have been designed to ease the task of developers: user interface builders and application frameworks.

User-interface builders

A user interface builder allows the developer of an interactive system to create the presentation of the user interface, i.e. the tree of widgets, interactively with a graphical editor. The editor features a 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.

User interface builders focus on the presentation of the interface. They also offer some facilities to describe the behavior of the interface and to test the interaction. Some systems allow the interactive specification of common behaviors such as a menu command opening a dialog box, a button closing a dialog box, a scrollbar controlling a list or text. The user interface builder can then be switched to a "test" mode where widgets are not passive objects but work for real. This may be enough to test prototypes for simple applications, even though there is no functional core nor application data.

In order to create an actual application, the part of the interface generated by the UI builder must be assembled with the missing parts, i.e. the functional core, the application interface code that could not be described from within the builder, and the run-time module of the generator. Most generators save the interface into a file that can be loaded at run-time by the generator's run-time (Fig. 20). With this method, the application needs only be re-generated when the functional core changes, *not* when the user interface changes. This makes it easy to test alternative designs or to iteratively create the interface: each time a new version of the interface is created, it can be readily tested by re-running the application.

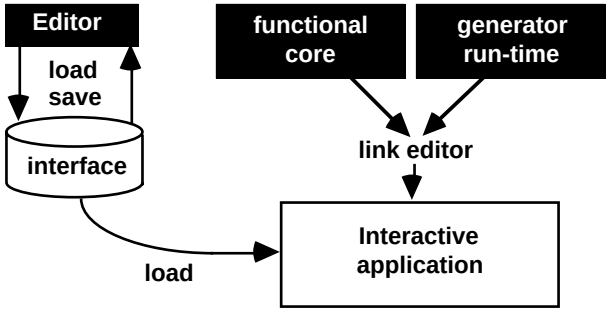

Figure 20: iterative user interface builder.

In order to make it even easier to modify the interface and test the effects with the real functional core, the interface editor can be built into the target application (Fig. 21). Changes to the interface can then be made from within the application and tested without re-running it. This situation occurs most often with interface builders based on an interpreted language (e.g. Tcl/Tk, Visual Basic).

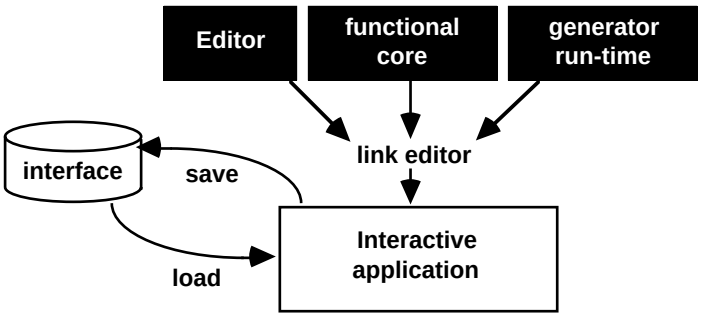

Figure 21: interactive user interface builder.

In either case, a final application can be created by compiling the interface generated by the user interface builder into actual code, linked with the functional core and a minimal run-time module. In this situation, the interface is not loaded from a file but directly created by the compiled code (Fig. 22). This is both faster and eliminates the need for a separate interface description file.

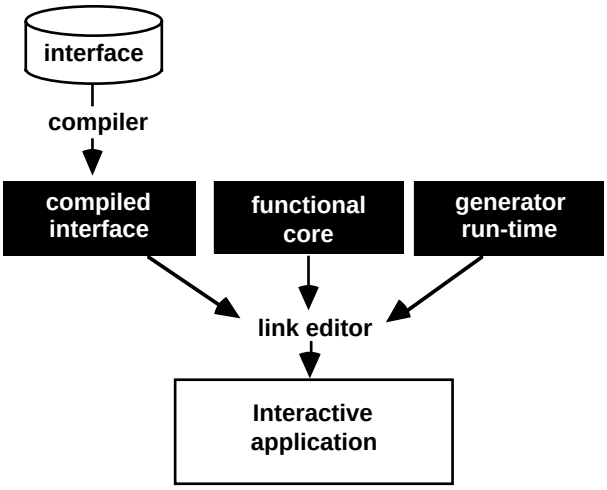

Figure 22: generation of the final application.

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, only the presentation. Therefore they still require a significant amount of programming, namely some part of the behavior and all the application interface. Systems such as NeXT's Interface Builder (Next, 1991) ease this task by supporting part of the specification of the application objects and their links with the user interface. Still, user interface builders require knowledge of the underlying toolkit and an understanding of their limits, especially when prototyping novel visualization and interaction techniques.

5.2 Software environments

Application frameworks

Application frameworks address a different problem than user interface builders and are actually complementary. 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, 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 with the functional core and the actual presentation of the non-standard parts of the interface. Most frameworks have been inspired by MacApp, a framework developed in the eighties to develop applications for the Macintosh (Apple Computer, 1996). Typical base classes of MacApp include Document, View, Command and Application. MacApp supports multiple document windows, multiple views of a document, cut/copy/paste, undo, saving documents to files, scripting, and more.

With the advent of object-oriented technology, most application frameworks are implemented as collections of classes. Some classes provide services such as help or drag-and-drop and are used as client classes. Many classes are meant to be derived in order to add the application functionality through inheritance rather than by changing the actual code of the framework. This makes it easy to support successive versions of the framework and limits the risks of breaking existing code. Some frameworks are more specialized than MacApp. For example, Unidraw (Vlissides and 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 design 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.

Model-based tools

User interface builders and application frameworks approach the development of interactive applications through the presentation side: first the presentation is built, then behavior, i.e., interaction, is added, finally the interface is connected to the functional core. Model-based tools take the other approach, starting with the functional core and domain objects, and working their way towards the user interface and the presentation (Szekely et al., 1992, 1993). The motivation for this approach is that the *raison d’être* of a user interface is the application data and functions that will be accessed by the user. Therefore it is important to start with the domain objects and related functions and derive the interface from them. The goal of these tools is to provide a semi-automatic generation of the user interface from the high-level specifications, including specification of the domain objects and functions, specification of user tasks, specification of presentation and interaction styles.

Despite significant efforts, the model-based approach is still in the realm of research: no commercial tool exists yet. By attempting to define an interface declaratively, model-based tools rely on a knowledge base of user interface design to be used by the generation tools that transform the specifications into an actual interface. In other words, they attempt to do what designers do when they iteratively and painstakingly create an interactive system. This approach can probably work for well-defined problems with well-known solutions, i.e. families of interfaces that address similar problems. For example, it may be the case that interfaces for Management Information Systems (MIS) could be created with model-based tools because these interfaces are fairly similar and well understood.

In their current form, model-based tools may be useful to create early horizontal or task-based prototypes. In particular they can be used to generate a "default" interface that can serve as a starting point for iterative design. Future systems may be more flexible and therefore usable for other types of prototypes.

User interface development environments

Like model-based tools, user interface development environments (UIDE) attempt to support the development of the whole interactive system. The approach is more pragmatic than the model-based approach however. It consists in assembling a number of tools into an environment where different aspects of an interactive system can be specified and generated separately.

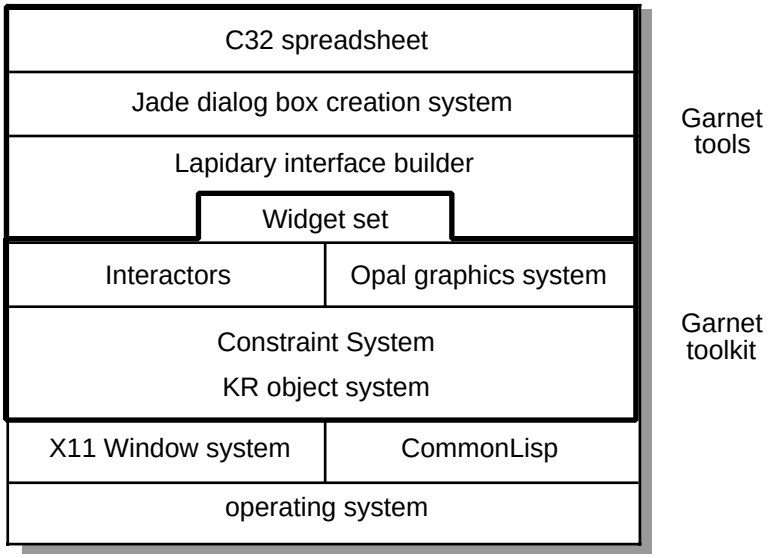

Fig. 23: The Garnet toolkit and tools (Myers et al., 1990)

Garnet (Fig. 23) and its successor Amulet (Myers et al, 1997) provide a comprehensive set of tools, including a traditional user interface builder, a semi-automatic tool for generating dialog boxes, a user interface builder based on a demonstration approach, etc. One particular tool, Silk, is aimed explicitly at prototyping user interfaces.

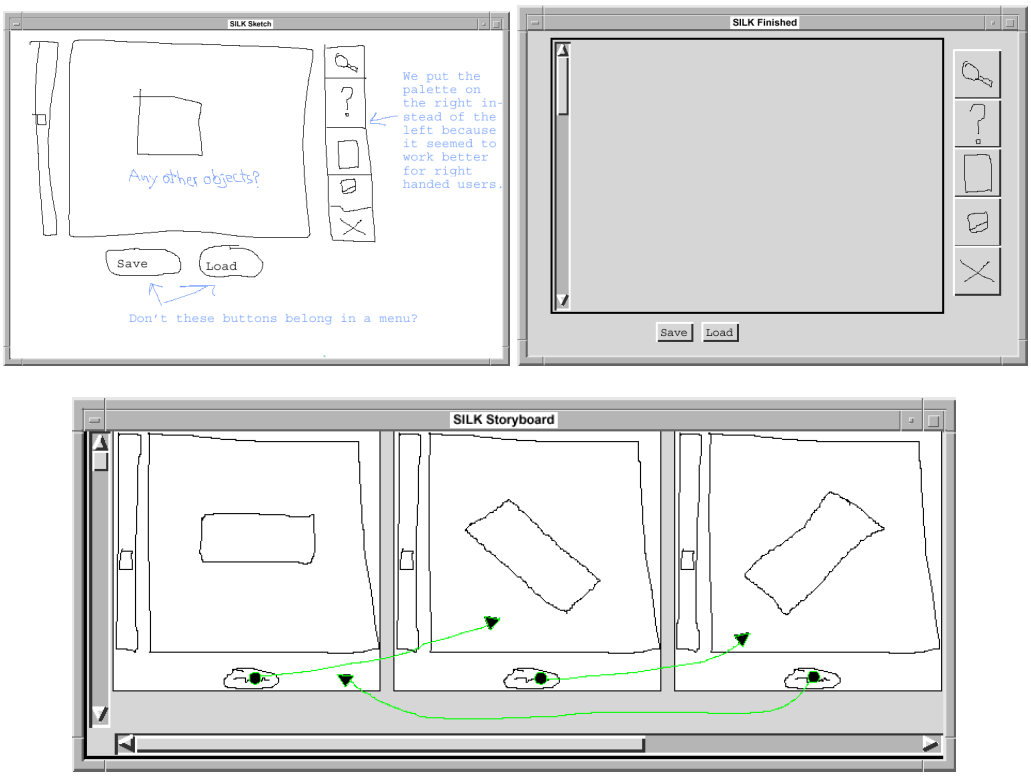

Fig. 24: A sketch created with Silk (top left) and its automatic transformation into a Motif user interface (top right). A storyboard (bottom) used to test sequences of interactions, here a button that rotates an object. (J. Landay, with permission)

Silk (Landay & Myers, 2001) is a tool aimed at the early stages of design, when interfaces are sketched rather than prototyped in software. Using Silk, a user can sketch a user interface directly on the screen (Fig. 24). Using gesture recognition, Silk interprets the marks as widgets, annotations, etc. Even in its sketched form, the user interface is functional: buttons can be pressed, tools can be selected in a toolbar, etc. The sketch can also be turned into an actual interface, e.g. using the Motif toolkit. Finally, storyboards can be created to describe and test sequences of interactions. Silk therefore combines some aspects of off-line and on-line prototyping techniques, trying to get the best of both worlds. This illustrates a current trend in research where on-line tools attempt to support not only the development of the final system, but the whole design process.

6. Evolutionary Prototypes

Evolutionary prototypes are a special case of iterative prototypes, that are intended to evolve into the final system. Methodologies such as Extreme Programming (Beck, 2000) consist mostly in developing evolutionary prototypes.

Since prototypes are rarely robust nor complete, it is often impractical and sometimes dangerous to evolve them into the final system. Designers must think carefully about the underlying *software architecture* of the prototype, and developers should use well-documented *design patterns* to implement them.

6.1 Software architectures

The definition of the software architecture is traditionally done after the functional specification is written, but before coding starts. The designers design on the structure of the application and how functions will be implemented by software

modules. The software architecture is the assignment of functions to modules. Ideally, each function should be implemented by a single module and modules should have minimal dependencies among them. Poor architectures increase development costs (coding, testing and integration), lower maintainability, and reduce performance. An architecture designed to support prototyping and evolution is crucial to ensure that design alternatives can be tested with maximum flexibility and at a reasonable cost.

Seeheim and Arch

The first generic architecture for interactive systems was devised at a workshop in Seeheim (Germany) in 1985 and is known as the Seeheim model (Pfaff, 1985). It separates the interactive application into a *user interface* and a *functional core* (then called "application", because the user interface was seen as adding a "coat of paint" on top of an existing application). The user interface is made of three modules: the presentation, the dialogue controller, and the application interface (Fig. 25). The *presentation* deals with capturing user's input at a low level (often called lexical level by comparison with the lexical, syntactic and semantic levels of a compiler). The presentation is also responsible for generating output to the user, usually as visual display. The *dialogue controller* assembles the user input into commands (a.k.a. syntactic level), provides some immediate feedback for the action being carried out, such as an elastic rubber line, and detects errors. Finally, the *application interface* interprets the commands into calls to the functional core (a.k.a. semantic level). It also interprets the results of these calls and turns them into output to be presented to the user.

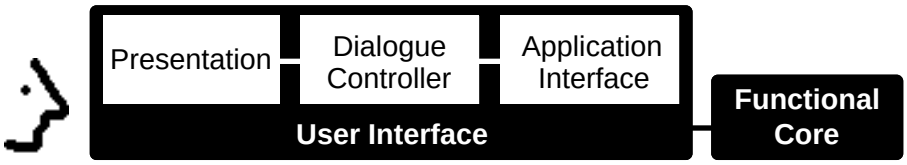

Figure 25: Seeheim model (Pfaff, 1985)

All architecture models for interactive systems are based on the Seeheim model. They all recognize that there is a part of the system devoted to capturing user actions and presenting output (the presentation) and another part devoted to the functional core (the computational part of the application). In between are one or more modules that transform user actions into functional calls, and application data (including results from functional calls) into user output.

A modern version of the Seeheim model is the Arch model (The UIMS Workshop Tool Developers, 1992). The Arch model is made of five components (Fig. 26). The *interface toolkit component* is a pre-existing library that provides low-level services such as buttons, menus, etc. The *presentation component* provides a level of abstraction over the user interface toolkit. Typically, it implements interaction and visualization techniques that are not already supported by the interface toolkit. It may also provide platform independence by supporting different toolkits. The *functional core component* implements the functionality of the system. In some cases it is already existent and cannot be changed. The *domain adapter component* provides additional services to the dialogue component that are not in the functional core. For example, if the functional core is a Unix-like file system and the user interface is a iconic interface similar to the Macintosh Finder, the domain adapter may provide the dialogue controller with a notification service so the presentation can be updated whenever a file is changed. Finally, the *dialogue component* is the keystone of the arch; it handles the translation between the user interface world and the domain world.

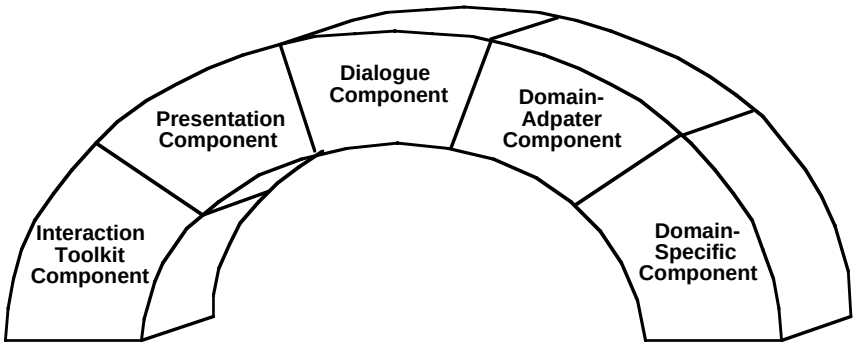

Figure 26: The Arch Model (The UIMS Workshop Developers Tool, 1992)

The Arch model is also known as the Slinky model because the relative sizes of the components may vary across applications as well as during the life of the software. For example, the presentation component may be almost empty if the interface toolkit provides all the necessary services, and be later expanded to support specific interaction or visualization techniques, or multiple platforms. Similarly, early prototypes may have a large domain adapter simulating the functional core of the final system, or interfacing to an early version of the functional core; the domain adapter may shrink to almost nothing when the final system is put together.

The separation that Seeheim, Arch and most other architecture models make between user interface and functional core is a good, pragmatic approach but it may cause problems in some cases. A typical problem is a performance penalty when the interface components (left leg) have to query the domain components (right leg) during an interaction such as drag-and-drop. For example, when dragging the icon of a file over the desktop, icons of folders and applications that can receive the file should highlight. Determining which icons to highlight is a semantic operation that depends on file types and other information and must therefore be carried out by the functional core or domain adapter. If drag-and-drop is implemented in the user interface toolkit, this means that each time the cursor goes over a new icon, up to four modules have to be traversed once by the query and once by the reply to find out whether or not to highlight the icon. This is both complicated and slow. A solution to this problem, called *semantic delegation*, involves shifting, in the architecture, some functions such as matching files for drag-and-drop from the domain leg into the dialogue or presentation component. This may solve the efficiency problem, but at the cost of an added complexity especially when maintaining or evolving the system, because it creates dependencies between modules that should otherwise be independent.

MVC and PAC

Architecture models such as Seeheim and Arch are abstract models and are thus rather imprecise. They deal with *categories* of modules such as presentation or dialogue, when in an actual architecture several modules will deal with presentation and several others with dialogue.

The Model-View-Controller or MVC model (Krasner and Pope, 1988) is much more concrete. MVC was created for the implementation of the Smalltalk-80 environment (Goldberg & Robson, 1983) and is implemented as a set of Smalltalk classes. The model describes the interface of an application as a collection of triplets of objects. Each triplet contains a model, a view and a controller. A *Model* represents information that needs to be represented and interacted with. It is controlled by applications objects. A *View* displays the information in a model in a certain way. A *Controller* interprets user input on the

view and transforms it into changes in the model. When a model changes it notifies its view so the display can be updated.

Views and controllers are tightly coupled and sometimes implemented as a single object. A model is abstract when it has no view and no controller. It is non-interactive if it has a view but no controller. The MVC triplets are usually composed into a tree, e.g. an abstract model represents the whole interface, it has several components that are themselves models such as the menu bar, the document windows, etc., all the way down to individual interface elements such as buttons and scrollbars. MVC supports multiple views fairly easily: the views share a single model; when a controller modifies the model, all the views are notified and update their presentation.

The Presentation-Abstraction-Control model, or PAC (Coutaz, 1987) is close to MVC. Like MVC, an architecture based on PAC is made of a set of objects, called PAC agents, organized in a tree. A PAC agent has three facets: the *Presentation* takes care of capturing user input and generating output; the *Abstraction* holds the application data, like a Model in MVC; the *Control* facet manages the communication between the abstraction and presentation facets of the agent, and with sub-agents and super-agents in the tree. Like MVC, multiple views are easily supported. Unlike MVC, PAC is an abstract model, i.e. there is no reference implementation.

A variant of MVC, called MVP (Model-View-Presenter), is very close to PAC and is used in ObjectArts' Dolphin Smalltalk. Other architecture models have been created for specific purposes such as groupware (Dewan, 1999) or graphical applications (Fekete and Beaudouin-Lafon, 1996).

6.2 Design patterns

Architecture models such as Arch or PAC only address the overall design of interactive software. PAC is more fine-grained than Arch, and MVC is more concrete since it is based on an implementation. Still, a user interface developer has to address many issues in order to turn an architecture into a working system.

Design patterns have emerged in recent years as a way to capture effective solutions to recurrent software design problems. In their book, Gamma et al. (1995) present 23 patterns. It is interesting to note than many of these patterns come from interactive software, and most of them can be applied to the design of interactive systems. It is beyond the scope of this chapter to describe these patterns in detail. However it is interesting that most patterns for interactive systems are behavioral patterns, i.e. patterns that describe how to implement the control structure of the system.

Indeed, there is a battle for control in interactive software. In traditional, algorithmic software, the algorithm is in control and decides when to read input and write output. In interactive software, the user interface needs to be in control because user input should drive the system’s reactions. Unfortunately, more often than not, the functional core also needs to be in control. This is especially common when creating user interfaces for legacy applications. In the Seeheim and Arch models, it is often believed that control is located in the dialog controller when in fact these architecture models do not explicitly address the issue of control. In MVC, the three basic classes Model, View and Controller implement a sophisticated protocol to ensure that user input is taken into account in a timely manner and that changes to a model are properly reflected in the view (or views). Some authors actually describe MVC as a design pattern, not an architecture. In fact it is both: the inner workings of the three basic classes is a pattern, but the

decomposition of the application into a set of MVC triplets is an architectural issue.

It is now widely accepted that interactive software is event-driven, i.e. the execution is driven by the user's actions, leading to a control localized in the user interface components. Design patterns such as Command, Chain of Responsibility, Mediator, and Observer (Gamma et al., 1995) are especially useful to implement the transformation of low-level user event into higher-level commands, to find out which object in the architecture responds to the command, and to propagate the changes produced by a command from internal objects of the functional core to user interface objects.

Using design patterns to implement an interactive systems not only saves time, it also makes the system more open to changes and easier to maintain. Therefore software prototypes should be implemented by experienced developers who know their pattern language and who understand the need for flexibility and evolution.

7. Summary

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 and 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 multi-disciplinary 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 begin by defining prototypes and then discuss them as design artifacts. We introduce four dimensions by which they can be analyzed: representation, precision, interactivity and evolution. We then discuss the role of prototyping within the design process and explain the concept of creating, exploring and modifying a design space. We briefly describe techniques for generating new ideas, to expand the design space, and techniques for choosing among design alternatives, to contract the design space.

We describe a variety of rapid prototyping techniques for exploring ideas quickly and inexpensively in the early stages of design, including off-line techniques (from paper&pencil to video) and on-line techniques (from fixed to interactive simulations). We then describe iterative prototyping techniques for working out the details of the on-line interaction, including software development tools and software environments We conclude with evolutionary prototyping techniques, which are designed to evolve into the final software system, including a discussion of the underlying software architectures, design patterns and extreme programming.

This chapter has focused mostly on graphical user interfaces (GUIs) that run on traditional workstations. Such applications are dominant today, but this is changing as new devices are being introduced, from cell-phones and PDAs to

wall-size displays. New interaction styles are emerging, such as augmented reality, mixed reality and ubiquitous computing. Designing new interactive devices and the interactive software that runs on them is becoming ever more challenging: whether aimed at a wide audience or a small number of specialists, the hardware and software systems must be adapted to their contexts of use. The methods, tools and techniques presented in this chapter can easily be applied to these new applications.

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.

8. References

Apple Computer (1996). Programmer's Guide to MacApp.

Beaudouin-Lafon, M. (2000). Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. *Proceedings ACM Human Factors in Computing Systems, CHI'2000*, pp.446-453, ACM Press.

Beaudouin-Lafon, M. (2001). Novel Interaction Techniques for Overlapping Windows. Proceedings *of ACM Symposium on User Interface Software and Technology, UIST 2001*, CHI Letters 3(2), ACM Press. In Press.

Beaudouin-Lafon, M. and 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*, CHI Letters 2(2):191-190, ACM Press.

Beaudouin-Lafon, M. and Mackay, W. (2000) Reification, Polymorphism and Reuse: Three Principles for Designing Visual Interfaces. In *Proc. Conference on Advanced Visual Interfaces, AVI 2000*, Palermo, Italy, May 2000, p.102-109.

Beck, K. (2000). *Extreme Programming Explained*. New York: Addison-Wesley.

Bederson, B. and 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*, pp.17-26, ACM Press.

Bederson, B. and 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*, CHI Letters 2(2):171-180, ACM Press.

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, ACM Press.

Boehm, B. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61-72.

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

Chapanis, A. (1982) Man/Computer Research at Johns Hopkins, *Information Technology and Psychology: Prospects for the Future*. Kasschau, Lachman & Laughery (Eds.) Praeger Publishers, Third Houston Symposium, NY, NY.

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.

Coutaz, J. (1987). PAC, an Object Oriented Model for Dialog Design. In *Proceedings of INTERACT'87*, Bullinger, H.-J. and Shackel, B. (eds.), pp.431-436, Elsevier Science Publishers.

Dewan, P. (1999). Architectures for Collaborative Applications. In Beaudouin-Lafon, M. (ed.), *Computer-Supported Co-operative Work*, Trends in Software Series, Wiley, pp.169-193.

Djkstra-Erikson, E., Mackay, W.E. and Arnowitz, J. (March, 2001) Trialogue on Design of. ACM/Interactions, pp. 109-117.

Dourish, P. (1997). Accounting for System Behaviour: Representation, Reflection and Resourceful Action. In Kyng and Mathiassen (eds), *Computers and Design in Context*. Cambridge: MIT Press, pp.145-170.

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

Fekete, J-D. and Beaudouin-Lafon, M. (1996). Using the Multi-layer Model for Building Interactive Graphical Applications. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST'96, ACM Press, p. 109-118.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading MA: Addison Wesley.

Gibson, J. J. (1979). *The Ecological Approach to Visual Perception*. Boston: Houghton Mifflin.

Goldberg, A. and Robson, D. (1983). *Smalltalk--80: The language and its implementation*. Reading MA: Addison Wesley.

Goodman, D. (1987). *The Complete HyperCard Handbook*. New York: Bantam Books.

Greenbaum, J. and Kyng, M., eds (1991). *Design at Work: Cooperative Design of Computer Systems*. Hillsdale NJ: Lawrence Erlbaum Associates.

Houde, S. and Hill, C. (1997). What do Prototypes Prototype? In *Handbook of Human Computer Interaction 2ed complety revised*. North-Holland, pp.367-381.

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

Krasner, E.G. and Pope, S.T. (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, August/September 1988, pp.27-49.

Kurtenbach, G., Fitzmaurice, G., Baudel, T., Buxton. W. (1997). 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, ACM Press.

Landay, J. and Myers, B.A. (2001). Sketching Interfaces: Toward More Human Interface Design. *IEEE Computer*, 34(3):56-64.

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

Mackay, W.E. (1988) Video Prototyping: A technique for developing hypermedia systems. Demonstration in *Proceedings of CHI'88, Conference on Human Factors in Computing*, Washington, D.C.

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

Mackay, W.E. and Fayard, A-L. (1997) HCI, Natural Science and Design: A Framework for Triangulation Across Disciplines. *Proceedings of ACM DIS '97, Designing Interactive Systems*. Amsterdam, Pays-Bas: ACM/SIGCHI, pp.223-234.

Mackay, W.E. (2000) Video Techniques for Participatory Design: Observation, Brainstorming & Prototyping. *Tutorial Notes, CHI 2000, Human Factors in Computing Systems*. The Hague, the Netherlands. (148 pages) URL: www.lri.fr/~mackay/publications

Mackay, W., Ratzer, A. and 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, ACM Press.

Myers, B.A., Giuse, D.A., Dannenberg, R.B., Vander Zander, B., Kosbie, D.S., Pervin, E., Mickish, A., Marchal, P. (1990). Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 23(11):71-85.

Myers, B.A. (1991). Separating application code from toolkits: Eliminating the spaghetti of call-backs. *Proceedings of ACM SIGGRAPH Symposium on User Interface Software and Technology, UIST '91*, pp.211-220.

Myers, B.A. and Rosson, M.B. (1992). Survey on user interface programming. In *ACM Conference on Human Factors in Computing Systems, CHI'92*, pp.195-202, ACM Press.

Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimotivtski, A., Doane, P. (1997). The Amulet environment. *IEEE Transactions on Software Engineering*, 23(6):347 - 365.

NeXT Corporation (1991). *NeXT Interface Builder Reference Manual*. Redwood City, California.

Norman, D.A. and 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: Scribner's.

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

Perkins, R., Keller, D.S. and Ludolph, F (1997). Inventing the Lisa User Interface. *ACM Interactions*, 4(1):40-53.

Pfaff, G.P. and P. J. W. ten Hagen, P.J.W., eds (1985). *User Interface Management Systems*. Berlin: Springer.

Raskin, J. (2000). *The Humane Interface*. New York: Addison-Wesley.

Roseman, M. and Greenberg, S. (1999). Groupware Toolkits for Synchronous Work. In Beaudouin-Lafon, M. (ed.), *Computer-Supported Co-operative Work*, Trends in Software Series, Wiley, pp.135-168.

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

Schroeder, W., Martin, K., Lorensen, B. (1997). *The Visualization Toolkit*. Prentice Hall.

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.

Szekely, P., Luo, P. and Neches, R. (1992). Facilitating the Exploration of Interface Design Alternatives: The HUMANOID. *Proceedings of ACM Conference on Human Factors in Computing Systems, CHI'92*, pp.507-515.

Szekely, P., Luo, P. and Neches, R. (1993). Beyond Interface Builders: Model-based Interface Tools. *Proceedings of ACM/IFIP Conference on Human Factors in Computing Systems, INTERCHI'93*, pp.383-390.

The UIMS Workshop Tool Developers (1992). A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin*, 24(1):32-37.

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

Wegner, P. (1997). Why Interaction is More Powerful Than Algorithms. *Communications of the ACM*, 40(5):80-91.

Woo, M., Neider, J. and Davis, T. (1997) *OpenGL Programming Guide*, Reading MA: Addison-Wesley