During the 1990s, scenario-based software development techniques became increasingly prominent in software engineering and human-computer interaction. Anecdotal reports suggest that they are pervasive in contemporary software development practice. A study of 15 software companies in northern Europe reported that all were using various scenario-based techniques. Most of the projects used scenarios in the iterative development and analysis of prototypes. Scenarios were found to be especially useful in helping development teams reach sufficient agreement to enable work to proceed, and in ensuring consistency with predecessor systems. Two of the projects turned to scenario-based techniques after the standard software development methods they had been employing became too cumbersome. All of the projects surveyed emphasized the need for more comprehensive scenario-based methods (Woodhams, et al. 1998).

This introductory chapter discusses the problems in software development that motivate the use of scenario-based development methods. It also provides an overview of the scenario-based framework that forms the basis for the rest of the book and the case study that will be used to illustrate the techniques. We start with a brief example of scenario-based design, then shift to a historical survey of issues in software and usability engineering. We conclude with a discussion of the major phases and methods employed in scenario-based development.

1.1 Design by Scenario: Marissa’s Gravity Project

In the mid-1990s, the U.S. government initiated a broad program of research and development in computer networking called the National Information Infrastructure. One target of this program was public education. It was hoped that networking could facilitate collaborative, project-based classroom learning, improve community involvement in public schools, increase access of rural schools to educational opportunities, enhance gender equity in science and mathematics education, and reduce equipment costs and handling.
In response to this national initiative, we created a partnership between Virginia Tech and the local school system to explore the uses of computer networks in science classrooms. Our initial wish list pointed in many directions; we were excited about the potential of computer networks. But many of the people in the new partnership were not software developers. They were not familiar with networking tools and applications. They were teachers, school administrators, parents, and other citizens. We needed to reach a consensus about the problems we would address and the approach we would take. We needed everyone to understand and agree on a plan.

We started by writing a problem scenario. In this context, a scenario is simply a story about people carrying out an activity; a problem scenario is a story about how the problem domain as it exists prior to technology introduction. The problem scenario for this project (Figure 1.1) describes a hypothetical end-of-class situation in middle school science: Marissa has a question, but her question is not simple and well formed, and in any case time has run out. Will the question be in her mind tomorrow? Will it matter to her a day later? Has an opportunity to make science real for one child just been lost?

Scenario questions are a good way to raise questions. Reading the problem scenario makes you wonder about the other students and what they are thinking. It also makes you think about how science classes work. Why is it important that the class period has ended? Why does this mean Marissa loses access to the teacher? Why can't Marissa work with other students on the problem? It seems a shame to waste Marissa's curiosity and energy.

In parallel, we wrote a design scenario that conveyed a new vision for science learning (Figure 1.2). This scenario describes a different experience for Marissa, where she continues to pursue her questions after school. She uses an online environment to collaborate with other students, running simulation experiments, making visualizations of results, and writing a report. Of course,

- Marissa was not satisfied with her class today on gravitation and planetary motion. She is not certain whether smaller planets always move faster, or how a larger or denser sun would alter the possibilities for solar systems.
- She stays after class to speak with Ms. Gould, but she isn't able to pose these questions clearly, so Ms. Gould suggests that she re-read the text and promises more discussion tomorrow.

Figure 1.2 A design scenario describing our initial vision of a virtual school.

this is just one of many design scenarios that might have been written. We wrote the story to merge the vision of the National Information Infrastructure project with our own starting biases and interests. But any design scenario would have had the same effect—making the new ideas accessible to our partners from the community and the school system.

The full story of this educational technology project is a long one (Chen, Bossert, & Carroll 1997; Carroll et al. 1998; Carroll, Bossert, Neale, et al. 2000; Harrison, et al. 2000). We worked for six years developing a suite of collaboration tools called the Virtual School (Figure 1.3). The tools allowed students from different classrooms (and occasionally with help from community mentors) to work together on science projects. Sometimes they worked together in real time; at other times they worked on their own at different times.

This system development project led us in many directions. We had to confront problems of running video conferences across school system firewalls, and to create an architecture for collaborative software. We also had to develop and apply participatory design methods that allowed teachers and students to participate in the design of the collaborative software (Carroll, Chen, et al. 2000).
The current Virtual School consists of several integrated tools. The main window in Figure 1.3 upper left shows which members of a collaborating group are currently logged on, their shared notebooks, and recent project activity. It also is used to initiate text chat, video conferencing, and email. The central window shows a project notebook shared by all group members; the student is working on the Robot Pictures section. The student is placing an annotation in that section, soliciting comments from other group members. One of the students in the remote collaboration is working on the introduction (indicated by the lack on the section tab.

This brief history of the Virtual School project makes a critical point. The system we built is the system envisioned in the design scenario. Of course this is not shocking—the initial design is almost never the one implemented. What is remarkable is that we wasted no resources in this change of direction. We did not have to build and throw away the wrong system, and then start again and build the right one. In fact, we barely noticed that our design changed in fundamental ways. The reason for this is that we used scenarios to raise and evaluate ideas as they came up. Scenarios let everyone contribute to analysis and design, critiquing proposals as they were suggested, comparing them with current classroom practices, mocking them up on paper, and revising them. Scenarios are rich and evocative design representations, but they are also cheap and easy to revise.

1.2 Managing Software Development

Software development is more than writing code. It is the activity of creating software; it is a process guided by systematic methods and techniques. The process is challenging because software is very abstract and very flexible; it is difficult to "see," but easy to modify. It is easy to introduce changes that address a specific concern but also undermine the overall design.

In the 1960s, the challenges of software development became enormous. Third-generation computer hardware enabled new applications, leading to software systems of much greater size and complexity. The resulting failures—one overrun, late delivery, and ineffective and unreliable systems—were labeled the software crisis. This crisis led to the emergence of software engineering as a professional discipline.

The software crisis has not yet been resolved. In the succeeding decades, hardware costs continued to plummet, even as performance capabilities soared. Software development methods improved and greater attention was paid to software. Nevertheless, software engineering practices still lag behind the increasing scale and complexity of software systems. No single method or technology has yet provided a remedy for the software crisis (see the "No Silver Bullet" sidebar).

1.2.1 Software Engineering

Software engineering is founded on the ideas of structured programming (Mills 1971). Programmers first define the major structures of a software system—the database, the event handler, and the network server, and so on—and then recursively decompose (break down) each structure into substructures. The driving vision is to gain control over design activities by making the software development process explicit and systematic. An early and influential model for software engineering was the waterfall: software development is organized into a series of modular phases, beginning...
No Silver Bullet

Fred Brooks (1987) analyzed the perennial yearning for a software technology "silver bullet"—a single innovation that could make software development cost drop in the way hardware costs have. He argued that the essence of software is the abstract construction of data, relationships, and algorithms. Software is complex because these constructions are highly precise, uniquely detailed, and invariant under many representations. This inherent complexity is exacerbated by the fact that software is flexible. It can conform to its context of application, so it typically does, at the expense of undermining its own conceptual integrity; it can be changed and changed again without end, so it changes constantly.

Brooks discussed many breakthroughs in the history of software: high-level languages, time sharing, programming environments, and more recently, object-oriented programming, expert systems, and program verification. But he argued that these breakthroughs have addressed the management of software representations rather than the essence of software construction. High-level languages reduce complexity by allowing programmers to think in terms of problem constructs, instead of being distracted by machine implementations that are not directly relevant to data relations and algorithms. Time sharing addresses complexity by allowing programmers to continuously interact with problems, instead of being interrupted by batch processes. Programming environments make it easier to simultaneously use different programs, including standard tools. None of these technologies offers direct support for the development of better abstractions about data, relationships, or algorithms.

Brooks suggested that one key to the problems of software development is the lack of a clear design. He argued that requirements can never be specified completely, precisely, and correctly, and that prototyping is necessary to develop correct conceptual structure. Systems themselves must be grown through incremental development. Finally, the key to successful projects is identifying and supporting great software designers.

The analysis of functional requirements, and continuing through software design, implementation, testing, and maintenance. Each phase produces one or more documents that are handed off as a specification for the work of the next phase. For example, the software design phase produces a design specification that is used by the implementation phase, which in turn produces the program codes used in the integration phase. The waterfall is a wonderful management tool, because projects are organized, traced, and measured in terms of progress through the phases.

Separating software development into modular phases is a good idea. However, as we will see again and again in this textbook, software development processes are filled with tradeoffs. Multiple views of design ideas are common, and progress in part requires understanding and working with competing arguments. In this book, we use tradeoffs as a general organizing technique, to identify general issues with competing lines of argument. A major concern for any software development project is the identification, discussion, and resolution of tradeoffs (see "Tradeoffs in Design" sidebar).

One tradeoff associated with waterfall methods is that many requirements for a system cannot be anticipated and specified in advance (as in the Virtual School example; see Tradeoff 1.1). Unanticipated dependencies or entirely new requirements are often discovered only after clients interact with a running prototype of a system (Boehm 1991). This suggests that requirements analysis must begin at both the beginning and the end of the development process; in the extreme, this means that you cannot know what to build until you first build the wrong system. At the least, this tradeoff demands some degree of testing and iteration in software development.

Tradeoffs in Design

A tradeoff about design in general is that there is never a single correct answer. Some solutions will be better than others, and some may even seem exactly right at the time, but there are always alternatives. This makes design difficult, but also what makes it fun.

One way to acknowledge this is to keep track of all tradeoffs that come up in a design process. For instance, making progress quickly trades off with more thorough analysis and planning. Designing a sophisticated user interface (UI) trades off with time to complete the project. Building a system that is incredibly easy to learn will lead to frustration down the line for expert users. And so on. People on a design team are constantly raising alternatives and discarding tradeoffs. Sometimes they even collect precise data to help them choose among alternatives with competing positive and negative arguments.

In this book we emphasize the important role of tradeoffs when thinking about design possibilities. Much of what we know about usability and user interface design can be captured in terms of tradeoffs, and we rely on this format to organize models, theories, and guidelines of human-computer interaction. In the case study materials that accompany each chapter, we show how to analyze and track specific design issues as tradeoffs.

A main effect of writing down and working with tradeoffs in design is the creation of a design rationale—documentation showing why specific design ideas are accepted or rejected (Miller & Carroll 1995). Although writing down the design rationale is time consuming (like any documentation effort), it can be very valuable when developing and maintaining complex systems over a long period of time.
1.2.2 Prototyping and Iterative Development

The waterfall model offers a framework for addressing the software crisis, but it is seen by many as idealized or simplistic. A strict linear flow of design specifications is unlikely to work for systems of nontrivial scale and complexity. Indeed, studies of programming practice have documented that skilled developers deliberately undermine waterfall methods (DeGrace & Spafford 1990).

A complement to structured development is prototyping, where designers develop one or more operational models to demonstrate a design idea. A prototype implements ideas that are abstract, making them concrete, viewable, and testable. By turning a proposal into a prototype, designers can test it for usefulness, feasibility, or other project concerns. Prototypes can be built at many phases; they can be used to evaluate requirements, high-level design, detailed software design, and so on (Brooks 1995). The feedback provided by prototyping is used to guide further development, but importantly, it may also be used to transform or reject aspects of the design.

However, prototyping and iteration create their own problems (Tradoff 1.2): Process management can be difficult if the system evolves too rapidly to document and track the changes. Constant change can also undermine the integrity of a software design. It is difficult to schedule and deploy personnel in this paradigm, because the trajectory of the process unfolds over time, rather than being planned in advance.

One way to integrate prototyping with structured design is to view prototyping as a requirements analysis method—prototypes are built as trial versions of a system that will be later discarded (Brooks 1995). Another approach is to assume that prototyping will be used to review design documents. In iterative development, design documents are produced as an output of each phase, but they are continually refined through prototyping and testing. The final version of the design specification describes the system that was built (Carroll & Rosson 1985).

1.3 Usability in Software Development

Through the 1970s, it became clear that an important component of software engineering would be the user interface design. As more and more software was developed for interactive use, attention to the needs and preferences of end users intensified. In the 1960s, most software users were computing professionals who developed and maintained applications for their organizations. But as computing technology became more powerful, users became more diverse. Application installations and customization were simplified. Office professionals began to assume responsibility for operating their own applications. With personal computers (PCs), it became typical for end users to install and manage their own software. Today, the typical setting for software use is an ordinary person, or group of people, accessing information from the World Wide Web or using a software package installed on their PC.

1.3.1 The Emergence of Usability

As end users became more diverse and less technical, interactive systems came to be compared and evaluated with respect to usability—the quality of a system with respect to ease of learning, ease of use, and user satisfaction. The factors that make a system more or less usable are complex and are still the topic of considerable research. However, we can identify three distinct perspectives—human performance, learning and cognition, and collaborative activity—that have contributed to modern views of usability (Figure 1.4). Although these perspectives emerged at different points in time, they are not independent, and their relation to user satisfaction includes all three.

Human Performance

Usability entered the software development process at both ends: requirements and system testing. Marketing groups interviewed customers and analyzed competitive products to understand requirements. Quality assurance groups tested whether systems met design specifications regarding human performance with...
Chapter 1: Scenarios-Based Usability Engineering

1.3 Usability in Software Development

Figure 1.4: Three perspectives contributing to the general concept of usability.

Usability

- Human cognition, mental models of plans and actions
- Collaboration, group dynamics, and workplace context
- Human performance, time, and error

The scientific foundations for studies of human performance are psychology and industrial engineering. The emphasis is on optimal performance—simpler displays and commands, fewer keystrokes, and shorter execution times (Cali, Moray, & Newell 1980). Laboratory experiments are the norm, with results organized into guidelines for developers. A limitation of this approach is that the laboratory situations evaluated by researchers must often be simplified. Because users working with interactive systems exhibit great variability, researchers are often able to obtain statistically reliable results only by examining high-level comparisons (e.g., comparing menu versus command-line interaction).

It became clear that optimization of human performance would not be a comprehensive foundation for usability. There were several reasons for this. Although optimal performance is important in some circumstances, it is too narrow an objective. Users interfaces that are optimized for keystrokes are not necessarily easier or more satisfying in use. The reliance on formal laboratory experiments was also problematic. Setting up such studies is slow and expensive, and they often produce findings too general to be of practical value.

Nonetheless, the specification and testing of human performance objectives demonstrated that usability could be studied empirically, and that it could play a role in software development. A first generation of experimental methods was developed, and important design variables were identified (e.g., display complexity and length of command strings). The human performance perspective also initiated two enduring themes in usability: (1) the search for broader foundations in science, and (2) the search for more effective roles within the software development process.

Human-Computer Interaction

As the PC era dawned in the early 1980s, new usability challenges emerged. There was far greater variety among users and applications, and in the ways organizations were influenced by usability problems. Rapid learning and self-study became critical, and product development cycles were compressed. This placed a high premium on lightweight methods for improving system usability, including inspection methods based on guidelines or theory. New programming languages and tools were helping to streamline software development. Small and distributed software development organizations became common, and programming was used more and more to drive system development.

The increasing prominence of PCs in society made usability more visible. One important new user group consisted of cognitive scientists—psychologists.
anthropologists, sociologists, and philosophers interested in how people solve problems and learn new things. In the 1970s, many of these scientists were starting to use computers for their own research activities. Their personal experiences then prompted research programs exploring how people learn to use and solve problems on computers.

Text editing was the first computer-use experience for many of these researchers (e.g., Douglas & Moran 1983). Text editing is a complex task made up of lengthy sequences of planning, building, and modifying text structures. The goals and constraints of text editing change as text is created, text-editing skills develop over weeks and months, not just minutes; and writers possess real job skills, and accomplish real work. Many cognitive scientists felt that the field needed to study complex tasks of this sort. Soon spreadsheets, drawing programs, personal databases, and other applications were also in use. This new area of shared interest between computer science and cognitive science was called human-computer interaction (HCI).

A significant early HCI project was GOALS (goals, operators, methods, and selection rules Card, Moran, & Newell 1983). GOALS is used to analyse the goals, methods, and actions of routine human-computer interaction. This was an advance in human performance testing, because it addressed the mental activities that guide behavior.

Other cognitive scientists studied the learning challenges of interactive systems. Learning requires connecting two sorts of knowledge, the task and the computer. An important issue here is how task needs are mapped to application services. Another issue is the role of the user's prior knowledge—especially users try to understand computers by analogy to familiar activities and objects. This observation led to a variety of new user interface ideas, such as the now-pervasive "mneny" display, and user interactions based on the concept of direct manipulation (Chapters 3 to 5).

A second impact of HCI was on the activity of design itself. Empirical studies of software developers revealed that they are quite opportunistic in their problem solving: Programmers do not just analyze, design, and then implement software. Instead they work on pieces of a solution that seem tractable, and then reconsider their understanding of the entire problem (Gould 1990). The overall process is incremental and locally controlled.

These studies of design problem solving implied that usability input could be useful at any point in system development. As a result, usability evaluators began to use flexible and rich methods that could support a process of continual redesign. Thinking about techniques users comment on (as they work through tasks) were shown to be of great use in understanding the mental activities that guide computer use. Such techniques were applied within an iterative development framework that studied end users' reactions early and continually throughout development (Gould & Lewis 1985).

By the late 1980s, the concept of usability and its role in system development had changed profoundly. Usability was no longer seen as ensuring the quality of finished systems, or even as simply making functionality more accessible. It referred to a comprehensive process that included informal prototyping, thinking aloud evaluation, and regular user involvement in requirements analysis and design. In many development organizations, marketing analysts worked with usability engineers to develop requirements in an iterative fashion. These changes meant that requirements and designs could not be specified in advance.

One result was that software development activities became more unstructured and unpredictable.

Collaboration and Group Interaction

In the 1990s, the scope of usability broadened to incorporate social and organizational aspects of system development and use. In part this is due to the success of HCI. As usability specialists were asked to assist and direct a greater range of system development activities, their knowledge and skills expanded. In particular, as they played a larger role in requirements analysis, they began to address the social and organizational context in which people learn and use computers.

The broadening of usability was also caused by the internationalization of computer science in the 1980s. In European traditions, social and organizational aspects of human activity are less isolated from cognitive and behavioral aspects that they are in the U.S. (Kjeldsens, Ehn, & Kyng 1983). Finally, usability concepts and issues broadened in its response to the new technologies for communication and collaboration that swept through the computing industry and society.

Usability came to include more emphasis on understanding the activities of users in the real world (Buchanan 1987). This went beyond what is sometimes called task analysis; it involved detailed studies of work practices, roles, and concepts. Field methods were adapted from anthropology and sociology; usability engineers sometimes spent months at a work site collecting the data for requirements analysis. The descriptions such work produces are rich, but not very structured. The scenario-based framework presented in this textbook offers one approach to working with work-oriented methods of this sort.

As a worldwide community of usability engineers emerged, the field's scientific foundations became more diverse. For example, usability engineers became interested in activity theory, which describes how technology is just one
part of a complex mix of social factors that influence individuals operating within a community (Nardi 1996, Chapter 2). But while such frameworks lead to a rich understanding of usage situations, the descriptions they produce tend to be ambiguous, because of a strong dependence on language.

Through the 1990s, electronic mail became a pervasive communication tool, and other Internet tools such as newsgroups, multiplayer domains, and real-time chat became more accessible. Communication and the coordination of work using networking software had powerful effects on organizations. The persistent nature of users finding and creating information on a PC became fundamental to the portrait of groups working together in a variety of times and places.

Computer-supported cooperative work (CSCW) is the current frontier for usability concerns and methods. For example, in the next decade considerable effort will be directed at building rich descriptions of work that can help in writing precise specifications of design solutions. However, usability engineering practice—and correspondingly the view presented in this textbook—continues to incorporate each of the three perspectives discussed here, that is, human performance, human-computer interaction, and computer-supported cooperative work.

1.3.2 Usability Engineering

The term usability engineering was coined by usability professionals from Digital Equipment Corporation (Geisler et al. 1986). They used the term to refer to concepts and techniques for planning, achieving, and working objectives for system usability. The key idea is that measurable usability goals must be defined early in software development, and then repeatedly during development to ensure that they are achieved (Henderson 1984; Gail 1986).

From the start, usability engineering has relied on user interaction scenarios. For example, early proposals suggested that developers should track changes in performance times, errors, and user attitudes for specific task scenarios (Carroll & Rosson 1985). Such a process would allow developers to measure the impact of particular design changes on usability. Although the focus might be on the general implications of display features or commands, these functions would be considered in a concrete user interaction context.

Initially, usability engineering focused on the design of the user interface—on engineering effective interactive presentations of information and functions. More recently, the management of usability goals has been extended to other software development activities, particularly requirements analysis and system environment. This had a notable effect on software development documents. In the early 1980s, increased attention to usability led to the inclusion of user interaction scenarios as appendices in design specifications. By the late 1980s, such scenarios appeared at the front of these documents. Scenarios conveyed design concepts vividly and succinctly, and presented core functions within a meaningful usage context.

In the past 10 years, usability has become a central focus in software development. A 1992 survey of software developers reported that close to 50% of a system's software is devoted to user interface (Meyers & Rossow 1992). But as we have observed above, the scope of usability is actually less than user interface development. Today, usability engineers help to determine what functionality is necessary and appropriate, as well as how it should be presented.

Nevertheless, usability is not everything. Many issues and constraints that bear significantly on the development of usable software fall outside the scope of usability engineering, and are not addressed directly in this book. Examples include team formation, resource allocation, or dependencies on legacy systems. Such concerns may be addressed by the methods of software engineering, or by business activities such as resource planning and scheduling. However, such concerns can have an indirect influence on the final usability of an interactive software system. Table 1.1 lists a number of such concerns; these concerns are often referred to as nonfunctional requirements, because like usability, they form external constraints on a project that are not directly related to the function it will provide (Somerville 1992). For each concern we briefly indicate how it might influence the goals of usability engineering.

Ultimately, software development is driven by economics. Even in a research setting where systems are designed to demonstrate or explore new concepts, project managers must make wise choices about relative costs and benefits of design features (Buxton & Newby 1990; Kuhle 1991). As a simple example, a system that requires communication support can include email with considerably less cost than a collaborative virtual environment. Usability experts may argue persuasively for the latter, but the decision maker must consider usability benefits in light of many other constraints such as those summarized in Table 1.1. While we acknowledge this, the presentation in this book focuses more narrowly on usability issues—the manner in which a system supports users' work activities, the relative ease with which the system is learned and used, and the satisfaction that its users experience.

1.4 Scenario-Based Usability Engineering

Computers do more than just provide information and services for people to use. The design of computing systems is part of an ongoing cycle in which new technologies raise new opportunities for human activity, as people's tasks change in
1.4 Scenarios-Based Usability Engineering

### Table 1.1: Examples of other nonfunctional requirements in a system development project that might interact with usability concerns

<table>
<thead>
<tr>
<th>Issue</th>
<th>Potential Impacts on Usability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team membership</td>
<td>Failure to include usability experts on a team will limit attention to and resolution of usability concerns</td>
</tr>
<tr>
<td>Project size</td>
<td>Large complex projects are more difficult to coordinate and may lead to a focus on a few high-level, most critical usability issues, time limits well correspondingly limit the amount of iteration possible</td>
</tr>
<tr>
<td>Legacy systems</td>
<td>Prior commitment to existing hardware or software platforms may over constrain the design space, eliminating some options that would significantly improve usability</td>
</tr>
<tr>
<td>Portability</td>
<td>The need to build multiple compatible versions of a system may encourage development of a &quot;lowest common denominator&quot;</td>
</tr>
<tr>
<td>Reliability</td>
<td>A need for highly reliable systems may require a distributed or otherwise complex architecture that later is reflected in the user interface</td>
</tr>
<tr>
<td>Maintainability</td>
<td>A system intended for a long lifetime may be designed in a modular fashion that separates tasks and activities seen as highly interrelated by users</td>
</tr>
<tr>
<td>Software economics</td>
<td>The technology cost of a useful function or user interface technique may be too prohibitive to warrant its inclusion in the design</td>
</tr>
</tbody>
</table>

---

Response to these opportunities, new needs for technology arise (Carroll 2000). The basic argument behind scenario-based methods is that descriptions of people using technology are essential in discussing and analysing how the technology is (or could be) reshaping their activities. A secondary advantage is that scenario descriptions can be created before a system is built and its impact felt (Rosson, Maier, & Bødker 1995; Weindling, et al. 1998).

### 1.4.1 User Interaction Scenarios

A user interaction scenario is a story about people and their activities (Carroll & Rosson 1994). For example, suppose an accountent wishes to open a folder displayed on his screen in order to open and read a memo. However, the folder is covered by a budget spreadsheet that he also needs to use while reading the memo. The spreadsheet is so large that it nearly fills the display. The accountant pauses for several seconds, then resizes the spreadsheet, moves it partially out of the display, opens the folder, opens the memo, resizes and repositions the memo, and continues working.

This is about as routine a work scenario as one could imagine. Yet even this story conveys important information about window management and application switching. People need to coordinate information sources, to compare, copy, and integrate data from multiple applications; computer displays inevitably get cluttered and people must find and rearrange windows in these displays. Scenarios highlight goals that are suggested by the appearance and behavior of a system, what people try to do with the system: what procedures are adopted, not adopted, and carried out successfully or unsuccessfully; and what interpretations people make of what happens to them.

Scenarios have characteristic elements (Table 1.2). They include or presuppose a setting or starting state. The accountant scenario describes the starting state for the episode: the relative positions of the folder and spreadsheet and the presence of a single user. Other setting factors are implied by identifying the person as an accountant, and the work objects as budgets and memos.

Scenarios describe the behaviors and experiences of actors. There is just one actor in the example, but many human activities involve several or even many actors. Each actor has task goals. These are changes that the actor wishes to achieve in the circumstances of the setting. Every scenario involves at least one actor and at least one task goal. When multiple actors or goals are involved, some are usually more prominent than others. Often one goal is the highest-level scenario goal; this high-level goal answers the question "why did this story happen?" Similarly, one actor often has a principal role, the answer to the question "who is this story about?"

In the accountant scenario, the high-level goal is to compare the budget and memo information. A lower-level goal often called a subgoal is opening the folder that contains the memo; a further subgoal is resizing and moving the spreadsheet to expose the folder. Each goal or subgoal of a scenario is tied to efforts aimed at achieving the goal. Translating a goal into action and making sense of what happens afterward usually takes place inside the actor's mind. But sometimes this mental activity is important to a situation; so scenarios usually include information about planning and evaluation. The emphasis on people's changing goals, plans, and understandings is one thing that distinguishes user interaction scenarios from the related software engineering concept of use cases (see "Use Cases" sidebar).
Table 1.2 Characteristic elements of user interaction scenarios.

<table>
<thead>
<tr>
<th>Scenario Element</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setting</td>
<td>Situational details that motivate or explain goals, actions, and reactions of the actors</td>
<td>Office within an accounting organization, state of work area, tools, etc., at start of narrative</td>
</tr>
<tr>
<td>Actors</td>
<td>Human(s) interacting with the computer or other setting elements; personal characteristics relevant to scenario</td>
<td>Accountant using a spreadsheet package for the first time</td>
</tr>
<tr>
<td>Task goals</td>
<td>Effects on the situation that motivate actions carried out by actors</td>
<td>Need to compare budget data with values questioned in memo</td>
</tr>
<tr>
<td>Plans</td>
<td>Mental activity directed at converting a goal into a behavior</td>
<td>Opening the memo document will give access to memo information; resizing one window will make room for another</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Mental activity directed at interpreting features of the situation</td>
<td>A window that is too large can be hiding the window underneath; dark borders indicate a window is active</td>
</tr>
<tr>
<td>Actions</td>
<td>Observable behavior</td>
<td>Opening memo document; resizing and repositioning windows</td>
</tr>
<tr>
<td>Events</td>
<td>External actions or reactions produced by the computer or other features of the setting; some of these may be hidden to the actors but important to scenario</td>
<td>Window selection feedback; auditory or haptic feedback from keyboard or mouse; updated appearance of windows</td>
</tr>
</tbody>
</table>

Scenarios have a plot; they include sequences of actions and events, things that actors do, things that happen to them, changes in the setting, and so forth. These actions and events may aid, obstruct, or be irrelevant to goal achievement. Resizing and moving the spreadsheet supports the goal of opening the folder. Resizing and moving the spreadsheet in a way that enables simultaneous viewing with the spreadsheet. Pasting does not contribute directly to task goals, and the response depends on the result of that task. Each scenario can be seen as one instance of a use case. It specifies an execution thread for a particular starting state and set of events. Use cases play an important role in identifying software design articulates, services, and dependencies. They are also very useful in presenting and negotiating proposed functionality with clients.

Another difference between use cases and user interaction scenarios is the content included in the episode. Use cases are intended to be a complete description of what a system will do. User interaction scenarios specify functionality too, but always in the context of use. Scenarios focus less on completeness of coverage, directing attention instead to the design rationale and possible side effects of user-system interactions. Particularly early on in design, scenarios are deliberately underspecified, assuming that the details of a design will be worked out in an iterative process. One way to integrate the two methods is to develop use cases as a functional specification of user-system exchanges, and write scenarios that trace and consider the usability implications of these exchanges.

Although it suggests that the accountant's actions were not completely fluent or automatic.

Representing the use of a system or application with a set of user interaction scenarios makes the system's use explicit, and in doing so assists design and analysis toward a broader view of computers. It can help designers and analysts to focus attention on assumptions about people and their tasks. Scenario representations can be elaborated as prototypes, through the use of storyboards, videos, or rapid prototyping tools (Chapter 4).
1.4.2 Why Scenarios?

Design and engineering always involve the management of tradeoffs. For example, the focus in usability engineering is often on "eugenic"—optimizing user performance. However, usability engineers typically measure user satisfaction as well, and this does not always improve when performance is made more efficient (e.g., users may be required to learn complex keystroke combinations to speed interaction). Such a disparity between performance and satisfaction is not always problematic. Sometimes efficient performance is the highest priority goal, and as long as user satisfaction stays within specified limits, the product meets its usability goals. But if a team is trying to optimize both performance and satisfaction, difficult tradeoffs may arise.

Explicit usability objectives—such as a 20% reduction in performance times—are needed to guide the usability engineering process (Benwell, 1984; Gilb, 1984). They take an abstract usability goal such as "more efficient human performance" and make it concrete and testable. These usability objectives must be clear and specific to be effective. They should be related to what people know and can do and what they wish to accomplish. They should state what will indicate success and what experience a user should have when carrying out a set of task actions and measure performance.

At the same time, there is no guarantee that the starting goals of an engineering process are appropriate or even attainable. Sometimes good ideas are simply not practical. People may not accept the training needed to fully utilize a system's capabilities. The display hardware may not have the required resolution. Thus, while explicit and measurable goals are necessary, any particular goal may ultimately be discarded.

Scenario descriptions can be very useful in managing the tradeoffs of usability engineering (Figure 1.5). For example, scenarios are both concrete and flexible. A scenario can say what happens in a particular situation without committing to details of precisely how things happen. Much of the richness of a scenario is in the things that are not said. A scenario specifies its actors' goals and behaviors, and this can be arbitrarily detailed. Yet a scenario narrative is easily modified or elaborated, and can be made deliberately incomplete to help developers cope with uncertainty.

The concrete and flexible character of scenarios addresses the tension between wanting to make progress quickly but at the same time keeping the design space open for further change (Tradeoff 1.3). Designers can use scenarios to try out ideas and get feedback. But scenarios can be revised quickly and easily, helping to avoid premature commitment. Sharing and developing scenarios help to control the uncertainties of design work, while sharpening and strengthening design goals.

Another tradeoff in usability engineering is related to the observation that people's activities co-evolve with the technology they use. Designers must understand people's tasks to be able to support them with technology, but the new technology will change what people do (Tradeoff 1.4). For example, early spreadsheet programs revolutionized budget management. That success caused an increase in time spent using spreadsheet programs, which in turn caused...
spreadsheet users to develop new needs (Nielsen, et al. 1986). They wanted better support for budget projections, and better integration with other computer-based tasks, such as planning, communicating, and presenting information. Subsequent spreadsheet programs were designed to provide this support.

**Tradeoff 1.4**

Analyzing users' current tasks is essential in designing useful and usable systems, but new design change what people can do and how they choose to do it.

Scenarios help designers respond to current needs while also anticipating new needs. They offer insight into meaningful situations, but at the same time do not imply that things will stay the same. They describe systems in terms of the goals that people will be pursuing as they use the system. Scenarios focus designers on the needs and concerns of people in real world.

Most designers hope to facilitate human activity with elegant and innovative systems. Everyone wants to invent the replacement to the spreadsheet or a better Web authoring language. However, not all elegant and innovative ideas will succeed like the spreadsheet or HTML; even elegant and innovative functionality can sometimes undermine usability (Tradeoff 1.5).

**Tradeoff 1.5**

The rapidly evolving software market demands innovation and new features, but some functionality may actually undermine usability.

Sometimes good ideas are ahead of their time. In 1982, IBM produced the Audio Distribution System, a digital phone messaging system that even by today's standards had a very powerful set of file management and editing features. But the product did not do well in a marketplace oriented to analog phone answering technology—it was too advanced for the time. It required too much learning and too much conceptual change by users.

Software applications that are developed and refined over many versions often suffer from “creeping featuritis.” In searching for novel and elegant design ideas, developers can lose sight of people and their tasks. They push the technology envelope, but they do not create useful systems. Unfortunately, when functions do not contribute to the solution, they can become the problem: Unneeded functions create a learning burden for new users and continuing confusion for more experienced users.

Scenarios address this tradeoff by focusing software development on use, rather than on features that might enhance use. While it is certainly important for developers to push ahead with new technologies, pursuing them on a feature-by-feature basis invites side effects. Changing any individual user interface feature may impact the consistency of displays and controls throughout; it may even raise requirements for new functionality. Scenario-based analysis helps designers to steer between the twin risks of overconfidence attempting to accomplish too much and conservatism attempting to do too little.

Yet another tradeoff concerns communication and collaboration in software projects. Programmers often use technical design representations (e.g., a data flow diagram) to express and share ideas; managers use their own specialized representations (e.g., a job specifications) to describe their needs. These special-purpose representations increase the precision of communication. But they may also exclude participation by some individuals or groups who are important to successful development (Tradeoff 1.6). For example, end users are likely to have trouble following a data flow diagram; programmers may not understand the details or implications of a personnel description.

**Tradeoff 1.6**

Technical design representations can increase the precision of communication, but may exclude participation by untrained team members.

Scenarios address this tradeoff by using a universally accessible language. All project members can "speak" the language of scenarios. Scenarios facilitate participatory design—design work that takes place as a collaboration between developers and the people who will use the system (Muller 1991, 1992). Scenarios help to integrate many different kinds of knowledge and experience by simplifying communication among different kinds of experts. Within a development team, scenarios assist in handoff and coordination, by maintaining a guiding vision of the project's goals.

A final usability engineering tradeoff comes in the conflict between thinking and doing. Developers want to take action and make progress quickly, but too great a focus on software construction can work against the reflection and analysis needed to discover and implement high-quality design solutions (Tradeoff 1.7). Developers naturally reflect on their activities as they work. However, as humans we take pride not only in what we know and learn, but in what we can show at the end of the day. It is impossible to predict or understand everything in advance, and long discussions of alternatives can be frustrating. People want to act, to make decisions, to see progress.
1.5 Doing Scenario-Based Usability Engineering

This book shows how scenarios can be used to guide usability engineering—the scenario-based development (SBD) framework. Figure 1.6. The framework should not be understood as a waterfall, even though the diagram shows a "downward" flow from problem analysis to design and then to evaluation. At each step of the process, scenarios are analyzed and transformed in support of different development goals. We assume that all activities in SBD happen in an iterative and interlaced fashion, but for explanatory purposes we organize them into an idealized progression.

The chapters in the book motivate and illustrate the use of scenarios for addressing the many concerns of usability engineering. Together they demon-
1.5 Scenario-based Usability Engineering

information scenario. It is more than merely a matter of making it possible for people to see things on a computer display (Chapter 4).

The third substage involves the design of interaction scenarios. These scenarios describe the details of user action and feedback (Chapter 5). Each interaction scenario is a fully specified design vision: the users and tasks being supported, the information needed to carry out the task, the actions the users take to interact with the task information, and the responses the system provides to users' actions.

System design does not happen in a vacuum. Although the designers using SBD methods begin with problem scenarios, they must actively search for new ideas to use in transforming the problem scenarios into design scenarios. They draw on many resources in this—analyses of real-world situations and objects, and their own knowledge of current information technology, along with their understanding of human abilities and preferences and relevant design guidelines. Throughout the design process, claims analysis is used to identify and discuss key features and usability tradeoffs.

As with requirements analysis, the reflection and discussion promoted by scenario facilitate design at points where it is crucial to consider alternatives, and to work through the implications of design decisions. Scenarios are easy to write and change, so they support fluid and creative design reasoning. Again, as for requirements, the concrete and familiar form of scenarios enables input from many different interested parties.

1.5.3 Prototyping and Evaluation

SBD assumes that design ideas will be evaluated in a continuing fashion. This is often accomplished via a prototype that implements or demonstrates one or more pieces of the solution proposed in a scenario. Prototypes may be constructed at many points in design and with many different degrees of completeness or published (Chapter 6).

A prototype can take many forms. For example, a very rough sketch could be useful to prototype an activity scenario. Details of system interaction would not be specified, but people could evaluate the sketch in the context of the scenario. Prototypes can also take the form of a functional system, which can be used to test the design in a realistic setting. Prototypes can even be used to evaluate the design process itself, by allowing designers to see how their design ideas will work in practice.

In SBD we distinguish between formative evaluation, which is carried out to guide redesign, and summative evaluation, which serves a system verification function (Chapter 7). Questions involved in the latter are: Have we actually built...
the system that was envisioned and specified? Did we meet or exceed the usability goals quantified in the usability specifications? In product development, summative evaluation is sometimes called the "go/no-go" test. If the product fails in summative evaluation, the process may start over (perhaps with a new product manager).

In contrast, formative evaluation is aimed at improving a design prototype, not merely measuring overall quality. It asks questions such as: What is working poorly? Why? What changes might fix the problem? Why? By the time the development process carries out a summative evaluation, it may be sufficient to know simply whether goals have been met. But earlier in the process, the development team needs information that can guide further development.

Scenarios guide evaluation through usability specifications—user tasks with specified usability outcomes that are evaluated repeatedly to guide redesign work. Scenario narratives describe the actors in their environment, and other situation details that may influence people's ability to use or appreciate a prototype. Scenarios also predict the kinds of goals and reactions an evaluator will observe. As scenarios become more refined and concrete, they describe usability objectives more precisely, such as indicating how long a sequence of actions should take, or what reaction to a piece of system feedback should provoke. In this sense, a scenario can be seen as a test of hypotheses about system use.

1.5.4 Other Approaches

All software development methods—and this includes all usability engineering methods—struggle with the tension between a waterfall with well-specified handoffs, and a flexible prototyping approach. Resolving the tradeoff between these two perspectives will always lead to a linear development flow, accompanied by iterative feedback and refinement.

A good example is seen in Mayhew's (1999) usability lifecycle. This framework incorporates five major phases: requirements analysis, conceptual model design, screen design, detailed user interface design, and installation. Prototyping and iteration are integrated within a waterfall—all phases except requirements analysis include assessment activities that support iteration. The three central design phases (conceptual model design, screen design, and detailed user interface design) also include a combined assessment of whether all functionality has been addressed. If that assessment is not satisfactory, the process returns to requirements analysis.

Mayhew's usability lifecycle corresponds closely to the flow of analysis and development in SBD. One difference is in the role of scenarios in SBD as a unifying design representation. For Mayhew, the output of requirements analysis is a list of goals. The output of conceptual model design is a paper or computerized mock-up. In SBD, the output of any phase includes user interaction scenarios. A secondary contrast is that SBD emphasizes the documentation and reasoning about tradeoffs throughout development.

Other general approaches to usability engineering include Nielsen's (1992) work on heuristic evaluation, Boyce and Holzblatt's (1998) contextual design, and Constantine and Lockwood's (1999) essential use cases. As an eclectic framework, SBD shares features with all of these approaches: a mixture of analysis and design techniques, an emphasis on the detailed context of use, and an early emphasis on the basic functionality that users want. SBD differs from all of these other approaches in its central reliance on user interaction scenarios as an ongoing source of insight and reasoning about users' needs and experiences.

1.6 Example-Based Learning of SBD

Learning any method for software development is hard. Learning a usability method is especially hard, because human behavior is complex and unpredictable, and is influenced by so many interacting situation variables. Examples are very useful in presenting difficult material; a good example illustrates key concepts and can be used as a model for new problems (Gick & Holyoak 1980; Rickard 1984; Vardi 1984). Thus, we present SBD methods by example through a case study of usability engineering. The example is cumulative; it develops scenarios progressively from requirements analysis through evaluation.

1.6.1 Case Study: A Virtual Science Fair in MOOsburg

Our example is drawn from a community network research project. HCI researchers at Virginia Tech are working with the town of Blacksburg, Virginia, on a community network system called MOOsburg (see http://moozburg.cs.vt.edu). MOOsburg is a MOO—a collaborative environment modeled on a geographic space. In this case the town of Blacksburg. People use MOOsburg to post or review community information (as they might also do using conventional Web sites; see http://www.b Unett). However, MOOsburg also provides a variety of shared interactive tools such as chat, electronic whiteboard and notebook, message board, calendar, simulations, and so on (Carroll et al. 2001; 2001b). Co-present visitors can interact directly with these tools. Visitors can also create or modify objects for later use.
Figure 1.7 shows three people visiting a local history museum in MOOSburg. The main view displays a panorama of the real-world museum; using the field-of-view control at the upper left, visitors can rotate the image to see the entire room. In front of the image are icons representing the three people online at this location. The visitors are using the text chat tool to discuss some of the exhibits. On the right is a toolbox from which they can create other objects for collaborative activities—for example, a shared whiteboard or a message board.

The map at the lower right is used to navigate to the virtual museum. The map corresponds to the real-world map of Blackburg, and the virtual museum has been positioned at the real museum's street location in the town. MOOSburg users navigate by selecting dots on the map—each dot is an online site that can be visited—and places with other visitors are displayed in green. The map is zoomable, so that visitors can get an overview of the entire town (not the main

roads show at the highest level) or zoom into a very detailed view (all roads, streets, and buildings are displayed). In this screen shot the user has zoomed in enough to see the building outlines.

MOOSburg has been explored and extended through the development of specific activities and tools within the general collaborative framework. We chart the development of one of these activities—a virtual science fair (VBF)—as the case study. The virtual science fair uses MOOSburg tools (e.g., email, online chat, discussion forums, and multimedia displays) to create a supplement to a traditional American science fair.

In a science fair, students develop individual projects. They exhibit their projects at the fair, where the exhibits are judged and awarded prizes. Although a science project is developed over many months, the fair itself takes place over a few hours. Science fairs are open to the public, but normally are attended by the participating students and their friends and families. The high-level concept for the virtual science fair is to extend the boundaries of a traditional science fair.

Using MOOSburg, we hope to make the fair attractive and accessible to a larger group of community members, and to enrich the activities of exhibit construction, viewing, and judging.

The science fair case study will be presented in a cumulative fashion to illustrate the VBF process shown earlier in Figure 1.6. The central elements are covered in Chapters 2 through 8 (Table 1.3). Chapter 2 covers techniques for analyzing a problem situation (in this case a traditional science fair), and generating problem scenarios. Chapters 3 through 5 cover the three phases of design, first focusing on the activities that will be supported, then elaborating the scenarios to include the details of interaction representation and interaction techniques. Chapter 6 describes the creation of virtual science fair prototypes, and Chapter 7 illustrates how these prototypes can be used to evaluate usability. Chapter 8 illustrates the special design problems of creating effective learning and help materials.

Some of the case study materials presented in the text were developed in the MOOSburg project; others were created to serve the educational goals of this textbook. The result is a comprehensive model for students or practitioners who wish to apply scenario-based methods to usability engineering projects. At the same time, the science fair project has several domain-specific characteristics, including the following:

- there is no underlying business unit providing financial or organizational structure;
- it is relatively free of development schedules and similar external constraints;
Table 1.3: Scenario-based design techniques and methods illustrated by the virtual bicycle case study.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Techniques Illustrated</th>
<th>S/DB Analysis and Design Products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 2:</td>
<td>Stakeholder analysis, hierarchical task analysis, user profiles, interviews, artifact</td>
<td>Problem scenarios</td>
</tr>
<tr>
<td>Analyzing</td>
<td>analysis, videos, field notes, theme analysis, stakeholder diagrams, participatory</td>
<td>Claims documenting tradeoffs in stakeholders' current practices</td>
</tr>
<tr>
<td>Requirements</td>
<td>analysis</td>
<td></td>
</tr>
<tr>
<td>Chapter 3:</td>
<td>Exploration of conceptual metaphors and MOOmburg services, reasoning from problem</td>
<td>Activity design scenarios</td>
</tr>
<tr>
<td>Activity</td>
<td>claims, participatory design</td>
<td>Claims documenting tradeoffs in the design of VSF activities</td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 4:</td>
<td>Exploration of presentation metaphors and MOOmburg interaction technology, reasoning</td>
<td>Design scenarios elaborated to include interaction design details</td>
</tr>
<tr>
<td>Information</td>
<td>from activity claims, screen and icon design, participatory design</td>
<td>Claims documenting tradeoffs in VSF interaction design</td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 5:</td>
<td>Exploration of presentation metaphors and MOOmburg interaction technology, reasoning</td>
<td>Design scenarios elaborated to include interaction design details</td>
</tr>
<tr>
<td>Interaction</td>
<td>from activity and information claims, storyboards, participatory design</td>
<td>Claims documenting tradeoffs in VSF interaction design</td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 6:</td>
<td>Key screens, scenario machine, evolutionary development</td>
<td>Mock-ups, screens, and other prototypes of the interaction design scenarios</td>
</tr>
<tr>
<td>Prototyping</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 7:</td>
<td>Questionnaires, task instructions, data collection forms, verbal protocols and critical</td>
<td>Usability specifications</td>
</tr>
<tr>
<td>Usability</td>
<td>incident analysis, descriptive statistics</td>
<td>Usability test results</td>
</tr>
<tr>
<td>Evaluation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 8:</td>
<td>Exploration of learning and help metaphors and MOOmburg help facilities, reasoning</td>
<td>Documentation design scenarios</td>
</tr>
<tr>
<td>User</td>
<td>from activity, information, and interaction claims, storyboards, participatory design</td>
<td>Claims documenting tradeoffs in VSF documentation design</td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary and Review

This chapter has provided a brief review of the challenges and methods that have contributed to modern software engineering practices, and the emergence of usability engineering as a concern within software development. It has also introduced the basic concepts and rationale for scenario-based development. Central topics include:

- It is possible to combine prototyping with structured development—prototypes are used to evaluate, reconsider, and refine the specification documents produced during the different phases of development.
- Tradeoffs are a fundamental aspect of any design process and should be a central focus in design reasoning.
- Usability emerged as an issue when the population of end users and the situations in which computing was used expanded from specialty to general-purpose use.
- Usability can be seen as a union of three perspectives: the human factors of perception and performance, the cognitive interactions of humans and computers, and the groups and organizations that provide the context for personal computing.
- Usability engineering is supported by the specification and evaluation of measurable usability objectives throughout the system development lifecycle.
- Scenarios describe the setting, actors, and events of a user-computer interaction, but also include information about users' mental activity (goals, plans, and reactions).
- Scenarios are concrete descriptions of action that are rough and flexible in content and level of abstraction. These characteristics help usability.
Chapter 1   Scenario-Based Usability Engineering

Engineers address a number of fundamental tradeoffs in the design and development of usable systems.

- Scenario-based development can be seen as a modified waterfall where the analysis of requirements leads to design, and ultimately to testing and deployment.
- Scenarios integrate the many tasks of system development by first organizing the analysis of user needs, and then serving as central representations of user needs that are developed in a systematic manner through design, evaluation, and documentation activities.

Exercises

1. Do you agree with Brook's claim that there is no silver bullet? Why or why not?
2. Suppose you are involved in development of an online banking service. List the pros and cons of using only the waterfall model versus a process that includes rapid prototyping and iterative development.
3. Choose a common computer-based task domain (e.g., email, Web browsing, or word processing). Discuss one or two major tasks in this domain from the perspective of task and technology co-evolution. How have your tasks changed as a function of technology that has become available over the past few years? How has your own (or other users') needs influenced the technology?
4. Consider the design of your personal Web page. List some tradeoffs that you would consider (or did consider) in constructing this page.
5. Write a sample scenario that describes someone reading an email when she arrives at the office in the morning. Be sure to include all of the standard elements (Table 1.2).
6. Revise your scenario to consider the impact of different users, a different setting, or different goals or reactions.

Recommended Reading


Analyze the problem from the three perspectives on usability discussed in the chapter (Figure 1.4). Answer these questions:
- What online shopping issues are suggested by each perspective?
- How might you address these issues if you were to develop a sample application in this domain?
- How might Tradeoffs 1.3 through 1.7 apply to the process of developing an Internet shopping application?

After your initial discussions, sketch out a scenario-based process you could follow to develop an Internet grocery store, working from the summary of SBD (Figure 1.6). Show how the general activities summarized for SBD would be translated into your problem domain of food shopping.

Project Ideas

Organize into a small group (three to four students). Select a modern computing application domain that you are all familiar with, such as Internet shopping.