Chapter 2
Scenario-Based Usability Engineering

Digital Equipment Corporation was among the first software companies to employ usability engineering. In the development of the MicroVMS Workstation Software (VWS) for the VAXstation I, a set of measurable user performance goals guided the development process. A benchmark task was designed in which test subjects created, manipulated, and printed the contents of windows. The development team set a usability goal of reducing performance time by 20% between version 1 and version 2 of VWS. Measurements of actual user performance on various benchmark subtasks were made to identify areas of greatest potential impact in the design of version 1. These areas of great impact would be used to guide development of version 2. The team exceeded their goal, improving performance time on the benchmark task by 37%, while staying within the originally-allocated development resources. Interestingly however, measured user satisfaction for version 2 declined by 25% relative to version 1. (See Good, Spine, Whiteside and George, 1986).

What is Usability Engineering?

The term *usability engineering* was first used by usability professionals from Digital Equipment Corporation (Good, Spine, Whiteside, & George, 1986). The term was a label for a set of concepts and techniques that would help to systematically plan, achieve, and verify software usability. The key idea is to define usability goals early in software development — in terms of measurable criteria — and then assess them during development to ensure that they are either achieved or explicitly altered. As in the case of software engineering, the practice of usability engineering is largely empirical and often informal. It encompasses a wide variety of methods, including some with mutually incompatible foundations. For example, it is a matter of continuing debate whether usability engineering should be purely empirical, or whether it can develop general and reusable principles and foundations in science.

Even the earliest formulations of usability engineering incorporate a notion of user interaction scenarios. Good et al. (1986) identify papers by Bennett (1984), Butler (1985), Carroll and Rosson (1985), Gilb (1984) and Shackle (1984) as comprising the foundation of usability engineering. These works proposed and investigated methods in which performance times, errors and misconceptions, user attitudes, and other cognitive and behavioral attributes of specific task scenarios are tracked during system development. Such a process allows developers to ascertain the impact of particular design changes on usability as they consider those changes. Although the developers are concerned with the general design implications of display features or commands, they see these functions in the context of concrete user interaction scenarios.

The early formulations of usability engineering focused on user interface design, that is, on engineering effective presentations of information and functions to users. In the later 1980s, the argument for usability goals that are defined early, explicitly managed, and verified by measurements, was extended to other software development activities, especially to requirements analysis and system envisionment. This had a noticeable effect on software design documents: In the early 1980s, increased attention to HCI had led to the inclusion of user interaction scenarios as appendices in design documents. By the late 1980s, such scenarios appeared at the front of these documents. They were used to vividly and succinctly convey the design concept, and to present the core functions within a meaningful usage context. Carroll and Rosson (1990) suggested that such scenarios be considered a functional specification in prototyping-oriented software development.

Usability Specifications as an Engineering Tool

In the early 1980s, as interactive computing became more pervasive, a new wrinkle in the software crisis came into sharp focus. Even if correct software could be created rapidly enough to exploit new hardware and meet application needs, would that software be usable? The state of the art was embodied in Brook's (1975) slogan that in any software development process one has to "plan to thrown one away." This advice was generalized

into a method called “iterative design” (Gould & Lewis, 1985). In this method, developers are essentially advised to throw systems away until the users are happy. The problem is that something is needed to guide this process.

We invented the concept of usability specification to provide engineering guidance for iterative design (Carroll & Rosson, 1985); the concept was developed by analogy to functional specifications. Functional specifications list features that achieve task needs. For example, multiple windows might support a need to work in parallel with multiple data sets. Usability specifications list features that achieve usability goals. For example, being able figure out how to reposition a window in less than 10 seconds might meet an ease-of-learning requirement. In both cases, a key criterion is that the features in the specification are *verifiable*, i.e., subject to measurement and assessment.

The first examples of usability specifications focused on usability metrics. That is, the specification writing process took for granted that a word processor produces printed documents, that an operating system manages windows, and so forth. The concern was then to set specific testable targets for such tasks. More recently usability specifications have assumed a more central role in scenario-based design: Usability engineers not only set behavioral targets for the tasks supported by a system, they also design and evolve the user interaction scenarios that define what these tasks will be.

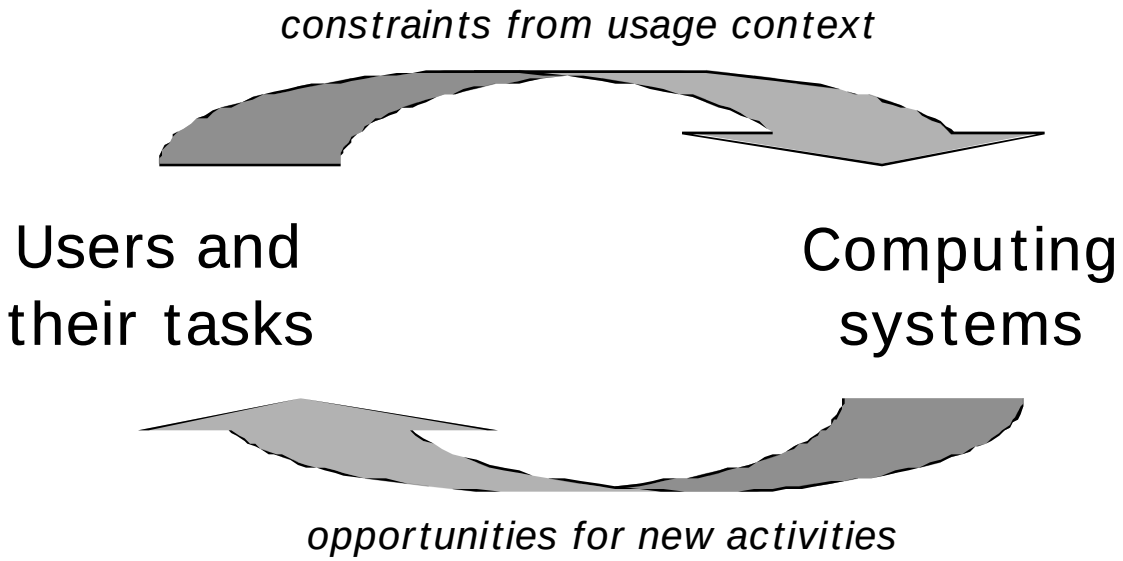

Figure 2.1: Co-evolution of user tasks and the systems that support them.

What are Scenarios?

Computers are more than just functionality. They unavoidably restructure human activities, creating new possibilities as well as new difficulties (Carroll, 1995; see Figure 2.1). At the same time, the usage contexts in which humans interact with computers provide detailed constraints for new applications of information technology. In analyzing

and designing interactive systems we need a better means to talk about how they may transform and/or be constrained by the context of human activity: this is the only way we can hope to attain control over the "materials" of design. A direct approach is to construct explicit descriptions of users' typical and significant activities early and throughout software development. Such descriptions — often called *scenarios* — support reasoning about situations of use, even before those situations are actually created.

Scenarios are stories. They are stories about people and their activities. For example, an accountant wishes to open a folder on the system desktop in order to access a memo on budgets. However, the folder is covered up by a budget spreadsheet that the accountant wishes to refer to while reading the memo. The spreadsheet is so large that it nearly fills the display. The accountant pauses for several seconds, resizes the spreadsheet, moves it partially out of the display, opens the folder, opens the memo, resizes and repositions the memo, and continues working.

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

Table 2.1: Characteristic Elements of User Interaction Scenarios

This is about as mundane a work scenario as one could imagine. Yet even this scenario conveys a vivid specification of 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; people need to find and rearrange windows in these displays. Scenarios highlight goals suggested by the appearance and behavior of the system, what people try to do with the system, what procedures are adopted, not adopted, carried out successfully or unsuccessfully, and what interpretations people make of what happens to them. If the accountant scenario is typical of what a user wants to do, it significantly constrains design approaches to window management and switching.

Scenarios have characteristic elements (see Table 2.1). They include or presuppose a *setting*: The accountant example explicitly describes a starting state for the described episode; the relative positions of the folder and spreadsheet, and the presence of the accountant. The scenario implies further setting elements by identifying the person as an accountant, and the work objects as budgets and memos.

Scenarios also include *agents* or *actors*: The accountant is the only agent in this example, but many human activities will include several or even many agents. Each agent or actor typically has task *goals* or *objectives*. These are changes that the agent wishes to achieve in the circumstances of the setting. Every scenario involves at least one agent and at least one goal. When more than one agent or goal is involved, they may be differentially prominent in the scenario. Often one goal is the defining goal of a scenario, the answer to the question "why did this story happen?" Similarly, one agent might be the principal actor, the answer to the question "who is this story about?"

In the accountant scenario, the defining goal is a comparison of budget and memo information. A sub-goal is opening the folder in which the memo is located, and a further sub-goal is moving the spreadsheet to allow the folder to be opened. Each goal or sub-goal of a scenario initiates a task directed at achieving the goal. Scenarios typically incorporate more than one task execution thread. The conversion of goals into intended actions or the interpretation of events with respect to task goals takes place in an agent's mind and is normally not observable. However when the details of a users' mental activity is important to a situation (e.g., for a new user), a scenario makes explicit reference to the *planning* and *evaluation* of goals.

Scenarios have a plot; they include sequences of *actions* and *events*, things that actors do, things that happen to them, changes in the circumstances of the setting, and so forth. Particular actions and events can facilitate, obstruct, or be irrelevant to given goals. Resizing the spreadsheet and moving it out of the display are actions that facilitate the goal of opening the folder. Resizing and repositioning the memo are actions that facilitate the goal of displaying the memo so that it can be examined with the budget. Pausing is an action that is irrelevant to any goal, though it suggests that this accountant's goal-directed actions were not completely fluent. Notably, actions and events can *change* the goals —

even the defining goal — of a scenario.

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 orients design and analysis toward a broader view of computers. It can help designers and analysts to focus attention on the assumptions about people and their tasks that may otherwise be implicit in systems and applications. Scenario representations can be elaborated as prototypes, through the use of storyboard, video, and rapid prototyping tools.

Why Scenario-Based Usability Engineering?

All engineering involves the management of tradeoffs. In Digital Equipment Company's VWS project (Good et al., 1986), the development team was focused on reducing users' performance time for the benchmark tasks. The team also measured user satisfaction, and found that it *declined* by almost 25% between the two versions. The VWS team did not regard this tradeoff as problematic, because they had also set a goal for user satisfaction that was exceeded in both versions. This is sound engineering. However, had the team's usability goal been to improve user performance *and* satisfaction by 20%, the situation would have been far more complicated.

Tradeoff 2.1: Explicit goals are necessary to manage a usability engineering process, BUT any given goal may turn out to be inappropriate or unattainable.

Explicit goals — such as a 20% performance time reduction — are needed to guide the usability engineering process. The goals need to be vivid and specific to be effective. The goals should be related to what the user knows and can do, what the user wishes to accomplish and what indicates success, and what the user experiences in carrying out various actions and obtaining various results.

However, there is no guarantee that the starting goals in any engineering process will be appropriate or attainable. Sometimes good ideas are simply not practical given the constraints of the problem: Users may not be willing or able to undergo enough training to fully utilize a system's capabilities. The display hardware may not be capable of presenting the resolution required. Thus although explicit and measurable goals are necessary for a usability engineering process, any particular goal may ultimately be discarded. Developers cannot make inflexible commitments to their goals, but neither can they be paralyzed by the uncertain future of the goals they set.

Scenario descriptions are useful in managing this tradeoff. They have the interesting property of being both *concrete* and *rough.* Scenarios can briefly sketch tasks without committing to details of precisely *how* the tasks will be carried out or *how* the system will enable the functionality for those tasks. They describe usability goals concretely by describing the situations in which the goals are pursued and achieved. They can be arbitrarily detailed. Yet, they are easily modified or elaborated, and can be made deliberately incomplete to help developers cope with uncertainty.

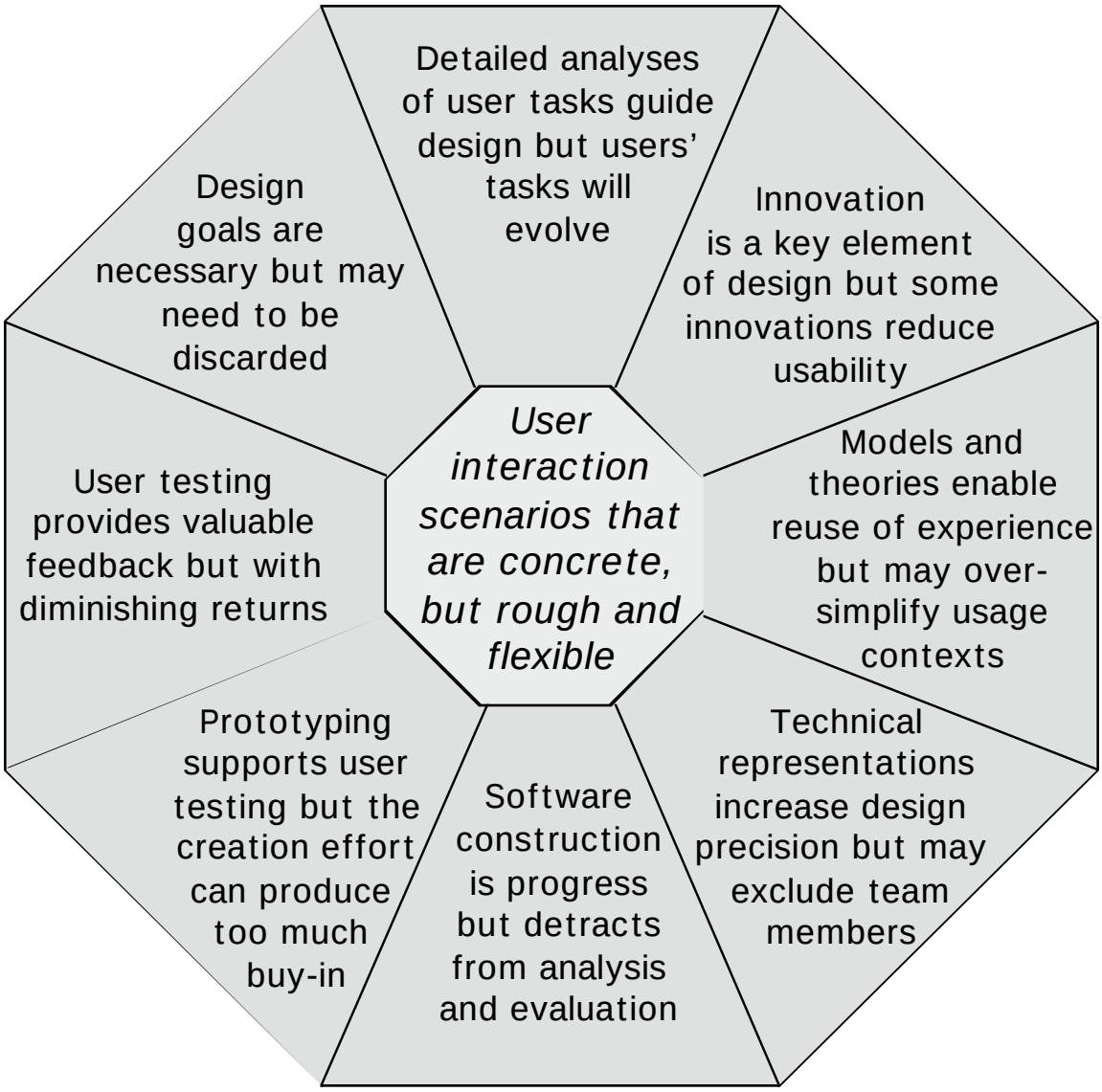

Figure 2.2: Fundamental tradeoffs in usability engineering addressed by scenarios.

Scenarios allow designers to make progress rapidly. They can be used to try ideas out and get directive feedback. But because they are flexible, they keep the design space open-ended and amenable to exploration, helping designers to avoid premature commitment. Much of their power comes from the way people create and understand stories. The human mind seems especially adept at gathering meaning from narratives, both in generation and interpretation, as illustrated by the remarkable examples of dreams (Freud, 1900) and myths (Levi-Strauss, 1967). Indeed, myths are universal tools for coping with uncertainty. Part of their cultural impact stems from being recalled and discussed throughout a community. On a more modest scale, design teams can do this too: Sharing and developing scenarios helps to control the uncertainties inherent in the work, while strengthening the team itself.

A second fundamental tradeoff in usability engineering comes from the fact that user requirements and technology co-evolve (Figure 2.1): We must understand user activity to be able to develop appropriate technology support for that activity, but the new technology will ultimately change what people do and how they choose to do it. For example, early spreadsheet programs revolutionized budget management. That success had the consequence that people spent lots of time using their spreadsheet programs, which in turn caused them to develop new needs (Nielsen & Mack, 1986). Spreadsheet users wanted better support for budget projections, such as natural order recalculation (in which cell dependencies determine the order of recalculation), and better integration with support for other tasks, such as planning, communicating, accessing information, reporting, and presenting. Subsequent spreadsheet program provided this support.

Tradeoff 2.2: We must understand users' tasks in order to develop usable technology, BUT new technology changes possibilities for action, and ultimately changes what people do and how they choose to do it

Scenario descriptions are useful in managing the tradeoff between responding to users' current tasks while anticipating new needs. By being concrete but rough, they give developers insight into meaningful human activity, but at the same time avoid the implication that things will remain the same. Scenarios capture the details of a work context, but they also emphasize the wide range of possibilities for the organization of work. They describe systems in terms of the work that people will try to do as they make use of those systems. A design process centered on scenarios is inherently focused on the needs and concerns of prospective users. Designers and clients are less likely to be captured by futuristic but inappropriate gadgets and gizmos — or to settle for routine technological solutions — when their discussions are couched in the language of scenarios.

One goal of most development projects is to facilitate human activity through providing elegant and innovative functionality, such as spreadsheet programs and graphical user interfaces. However, not all elegant and innovative functionality has this effect. A third tradeoff in usability engineering is that even functionality that is elegant and innovative can sometimes undermine usability.

Tradeoff 2.3: Developers and their organizations seek elegant and innovative functionality, BUT some functionality may actually undermine usability.

For example, an idea might be good but ahead of its 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. However, 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 of its users.

A more mundane generalization of this tradeoff is the well-known featuritis problem, discussed by Norman (1988) and many others. In their search for novel functions and elegant system concepts, developers often lose sight of the users and their tasks. They push the technology envelope, but they fail to create usable systems. When functions do not contribute to the "solution", they become the "problem": Unneeded functions increase the learning burden for new users, and are sources of continuing confusion for more experienced users.

Scenarios address this tradeoff by keeping the focus of software development work on use, not merely on the features and functions that might facilitate use. While it is clearly 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. A prominent current example is support for collaborative work. Making an editor collaborative allows many new sorts of user interactions; it supports a far richer variety of user tasks. However, it also creates many new usability problems, such as establishing and maintaining effective awareness of what one's collaborators have done to shared documents, what they are currently doing, and so on.

Working with a collaborative editing scenario, instead of a list of collaboration functions or modules, helps developers anticipate consequences and side-effects of new functionality, and design more comprehensive solutions. Scenario-based analysis helps organizations to steer between the twin risks of overconfidence (i.e., attempting to change or accomplish too much) and conservatism (i.e., attempting to do too little).

A fourth tradeoff in usability engineering pertains to the reuse of technical knowledge. Although each development project is unique, there is not enough time, effort, or money to develop unique solutions for each one. No one would imagine a need for unique operating system software for every installed machine; nowadays, no one would consider even creating unique applications for each installation. But the reuse of code in this sense is the trivial case. The greater usability engineering challenge is to reuse insights about users and their tasks, about the kinds of user interface displays and controls that are appealing and easy to learn, and about the kinds of functionality that people find tedious, distracting, or useless.

Tradeoff 2.4: Models and theories can help developers design usable systems, BUT abstractions oversimplify usability issues.

How can we characterize the lessons learned about interactive software such that they provide guidance to future efforts? Problem situations and solutions must be described at some level of abstraction if they are to be generalized. For example, perhaps a study has shown that automatic first-letter capitalization is undesirable in word processors. We could consider making this a design rule, but should we phrase it broadly (any text application) or narrowly (just personal word processors)? When exactly do we want to

suppress auto-capitalization? The tradeoff is that whereas models, theories, and other abstractions can help developers create usable systems, the abstractions oversimplify. It is dificult to tell just when they apply and how.

Scenarios can help to protect developers from oversimplifying by calling attention to contextual detail; they especially emphasize temporal dynamics and cause-and-effect relations in the situations they describe. Every element of a design, every move that a designer makes, has a variety of potential consequences. Scenarios provide a language for recognizing and addressing the many tradeoffs and dependencies in a design problem. They help developers think about the consequences of design changes — consequences for the software, for the tasks it supports, and the organizations in which the work takes place.

Scenarios provide a sort of middle-level abstraction that can be useful in drawing more general lessons. Scenarios can be classified in various ways and these classes can then be used to organize knowledge about design. Making a speech annotation on a video file is an example of creating meta-data to support later retrieval. Thus things we know about keyword indexing and retrieval might be useful whether the annotations were made via speech or something else. Conversely, things we learn about the use of speech annotations on video clips might be reused in other types of annotation scenarios.

A fifth tradeoff pertains to project-related communication and collaboration in the software development process. Software designers use technical representations such as data flow diagrams or state transition networks to express and share ideas; analysts develop diagrams such as organization charts or job descriptions to characterize an organization's internal structure and needs. Such representations increase communication precision. However, they may also *exclude* participation by some individuals or groups important to successful development. For example, users may be unable to follow a data flow diagram; programmers may not understand the details in a job description.

Tradeoff 2.5: Technical design representations can increase the precision of communication, BUT may exclude participation by some members of a project team.

Scenarios address this tradeoff by using a universally accessible language: all project members can "speak" the language of scenarios. Scenarios facilitate *participatory development* — design work that is a direct collaboration between developers and users. Enabling participation by users or other clients is a proactive approach to detecting and resolving organizational impacts that often result from inserting technology into a workplace. Scenarios can integrate diverse design skills and experience by simplifying communication among different kinds of experts. Among the development team, scenarios aid handoff and coordination by maintaining a guiding vision of the project's goals.

A sixth tradeoff in usability engineering comes from the conflict between *thinking* versus *doing*: thinking takes time and slows down action, but doing things gets in the way of thinking. Of course developers want to reflect on their activities, and they do so

routinely. However, people take pride not only in what they know and learn, but in what they can do and produce. They know from experience that it is impossible to predict or understand all potential outcomes and relationships, and they are often frustrated by discussions of alternatives. They want to act, to make decisions, to see progress.

Tradeoff 2.6: System-building moves a project forward, BUT can lead developers to direct too little effort toward analysis and evaluation.

Reflection about design activities is often de-coupled from the system's usage context. Design review meetings allow a project team to take stock of progress by working through objectives, progress reports, specifications, and so on. Such reviews facilitate design work in many ways, clarifying problems, alternatives, or decisions. However, a review removes designers from the day-to-day context of their work; the designers stop design work as such and reflect on interim results. Reviews do not evoke reflection *in the context of doing design.* Moreover, design reviews typically focus on listing and assessing features and functions, as opposed to considering a system's potential use.

The evocative nature of scenarios helps to address this. By depicting a concrete story of user interaction, a scenario vividly and succinctly conveys a vision of what the system will do. They stimulate imagination and invite designers to engage in "what-if" reasoning. In a scenario it is easy to change the values of several key variables at a time to imagine what new states or events might transpire. Designers can use scenarios to integrate their consideration of system requirements and the motivations and cognitive issues that underlie a user's actions and experiences. Scenarios can also describe designs at multiple levels of detail and with respect to multiple perspectives.

Two final tradeoffs concern evaluation. One of these concerns a hidden cost in implementing or prototyping early versions of a system. Although this is critical to iterative design — you must have something to test before you can get feedback — the time and effort spent in implementation can make it difficult to discard or seriously revise the design.

Tradeoff 2.7: Prototyping facilitates usability evaluation, BUT the time and effort to implement may make it difficult to discard a design based on usability evaluation.

A running system facilitates evaluation by providing something to evaluate. Of course it is possible to evaluate concepts and mock-ups; one can show drawings and task descriptions to users to get their reactions. But it is always better to have a demo or an executable prototype. Test users can experience the details of interaction, enter data, evaluate the execution of commands, and try to interpret displayed results. However, the more work one invests in a system, the more one values it. This well-known principle of psychology is called *cognitive dissonance* (Festinger, 1957). Cognitive dissonance is not neurotic. After all, if people fail to value something they spend weeks producing, what does that say about how they their own knowledge and skill? If we work hard, we should

be proud and committed to the result of our labor. However, cognitive dissonance can be a problem when we work hard and nevertheless our result must be abandoned or seriously reconsidered. Unfortunately, this happens frequently in software development.

Scenarios help designers manage cognitive dissonance in two ways. First, by being relatively lightweight, easy to create and work with, scenarios reduce the time and effort of developing something that can be evaluated. This reduces the cognitive dissonance evoked when evaluation data point to problems with the scenario. Second, scenarios stimulate the imagination; developing one scenario often evokes ideas for several others. Thus, even more than physical mock-ups and software prototypes, scenarios tend to be divergent, triggering many alternatives, and moderating allegiance to the original idea.

A final tradeoff comes from the need for a stopping rule in usability engineering. Evaluation guides iterative software development: As in the VWX case, problems are identified, prioritized, and addressed through redesign, and outcomes improve. The VWX case was quite simple; the team focused on improving overall performance on a single benchmark task. Contemporary usability engineering processes are often more complex. Usability engineers may consider a wide range of benchmark tasks and measure many parameters involving learning, performance, and satisfaction. Continuing cycles of evaluation and design change are expensive, and at some point the testing starts to produce smaller and smaller improvements in usability.

Tradeoff 2.8: Usability evaluation guides iterative development, BUT continuing design changes extend the development process (increased costs), and may not lead to noticable improvements in usability (diminishing returns).

The key, of course, is to anticipate diminishing returns, and avoid paying for the final cycles of iterative development. Scenarios help by providing an overall management tool in software development. The benchmark tasks for usability evaluation are simply testable user interaction scenarios — they are versions of scenarios used to envision and develop the system, but with metrics for user interaction specified as parameters for learning, performance, and attitude. Tracking the measured values of these parameters helps to determine whether evaluation has begun to produce diminishing returns.

A Scenario-Based Usability Engineering Process

This book introduces the simple framework for usability engineering shown in Figure 2.3. Scenarios are a central design representation and are passed among various software development activities. At each point, the scenarios are transformed in some way. In the following chapters, we will show how scenarios can be constructed and analyzed to do requirements analysis, to specify task models, to design information layouts and interaction sequences, to develop prototypes, to define usability specifications, and to write documentation.

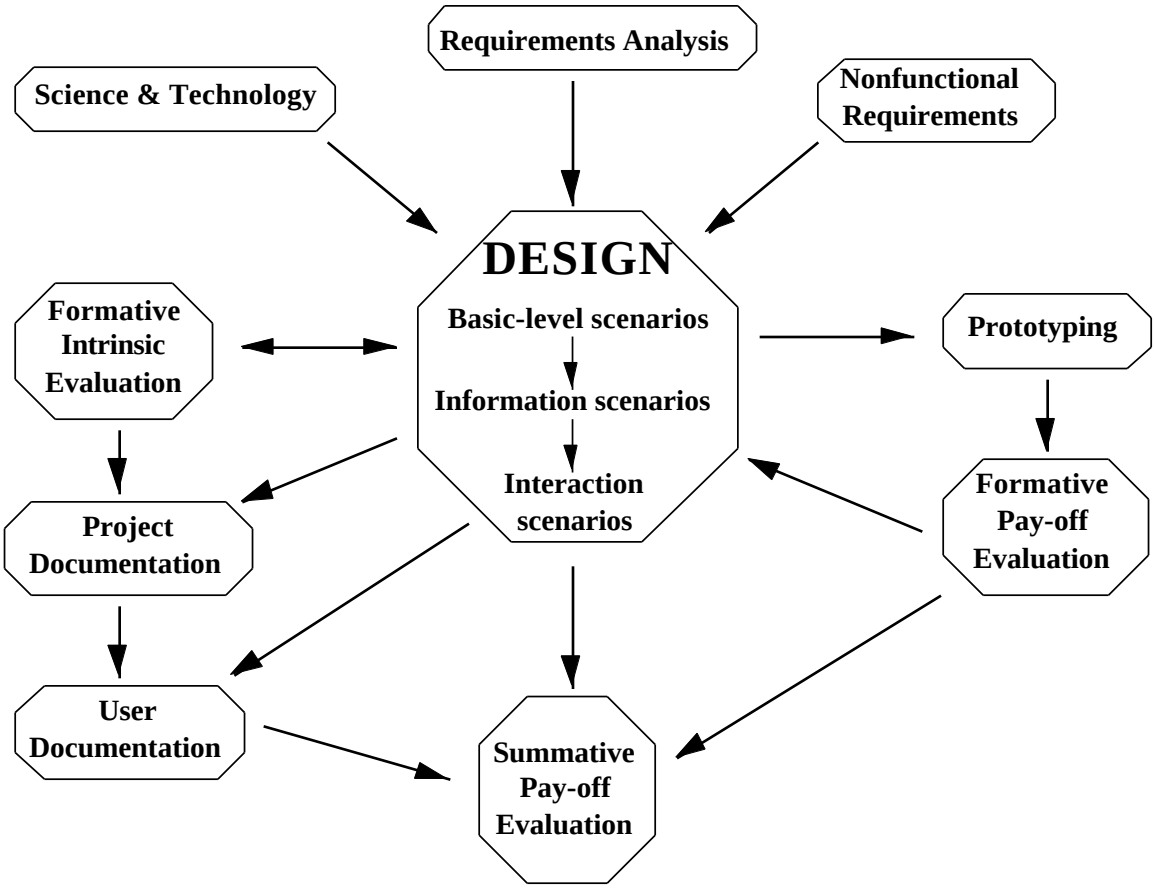

Figure 2.3: Overall organization of scenario-based usability engineering

Requirements

The flow of information and activity in scenario-based usability engineering (SBUE) echos the basic software development waterfall — the flow moves from requirements, through design, to evaluation. As observed In Chapter 1, a document-oriented waterfall process increases the coherence of the development. In SBUE, the documents passed among stages are scenarios. In requirements analysis, scenarios are gathered through interviews with clients and other users, through direct observations of user activity in the workplace, and through brainstorming among users and developers. The requirements scenarios describe the characteristics of the users, the typical and critical tasks they engage in, the artifacts they employ in their activity, and the organizational context of objectives and conventions (see Chapter 3).

In addition to the needs of users and their organizations, software development will be guided by the current state of science and technology, and by nonfunctional requirements. The current state of science and technology provides developers with a set of technical building blocks and constraints. The state-of-the-art specifies a level of technological refinement and sophistication that developers must take into account; normally, they hope to meet or exceed the state-of-the-art. In pushing the state-of-the-art,

developers can make use of new scientific results, or techniques and research prototypes recently demonstrated or described. They can apply or create design guidelines, derived from the state-of-the-art. And they can invent or modify system metaphors and conceptual models to guide the software development process.

Nonfunctional requirements include an assortment of issues that do not bear directly on system functionality, but that influence software development nonetheless. For example, the developers may be required to consider international or corporate standards regulating some system components (like keyboards and displays), restricting the design of some user interface elements, and defining the form of documents that must be produced in software development process. The system may need to be portable or even inter-operable across diverse hardware and software platforms, reliable to a specified criterion, or efficient with respect to memory and performance. The development team may have to operate within hard budgetary or schedule constraints.

Design

Design is the hub of the software development process. As a creative act, design can seem mysterious. Requirements analysis can be overwhelming, but at least it is anchored in the needs and possibilities of a real situation. Evaluation can also seem vast, but it too is anchored in the activities of real users working with concrete design ideas or prototypes. Design stands between two, vaguely indicating that a miracle occurs.

SBUE organizes design into three sub-stages. The sub-stages have a rough order: First, developers construct what basic-level scenarios — descriptions of the typical and critical activities that users will carry out with the software being developed. Such scenarios provide a concrete glimpse of the future that the designers are trying to enable through their efforts. These descriptions are *basic-level* because they describe user activity at the same level of detail that people use when they think about their own activity. One benefit of basic-level scenarios is that they increase the possibility that users can participate directly in the design process.

The second sub-stage is information scenarios. These are elaborations of basic-level scenarios, further detailing the information that users will provide to the system, as well as information that the system will provide in turn to users. There are many complexities to information scenarios. It is more than merely a matter of making it possible for users to provide information or to perceive information.

The third sub-stage is interaction scenarios. These scenarios describe the details of user action and feedback. At this point, the design scenarios contain all aspects of the design vision: the task(s) being supported, the information needed to carry out the task, and the actions required to interact with the task information.

Evaluation

Like requirements and design, evaluation is an immense topic. The culmination of evaluation, depicted at the bottom of Figure 2.3, involves a summative or “pay-off”

evaluation. This is a sort of system verification: It asks whether we have actually built the system that was described throughout the development process. It is the bottom line with respect to usability: Did we in fact build the system described in our scenarios? Did we meet or exceed the usability goals quantified in the usability specifications? Did we meet or exceed competitive evaluation results (that is, can we compete in the marketplace)? In product development, this stage of evaluation is sometimes called the "go/no go" test. If the product fails here, the process may start over (perhaps with a new product manager!).

Other kinds of evaluation take place throughout the development process. We called these "formative", following the terminology of Scriven (1967). Formative evaluation is directed at *improving* a prototype of design, not merely measuring it. By the time the development process gets to summative, pay-off evaluation, it may be sufficient to know simply that goals have been met or that they have not. One has reached a final decision point. But earlier in the process, the development team needs information that can guide further development.

Intrinsic formative evaluation involves analysis of properties and characteristics of the design. It does not include user testing ("pay-off" is Scriven's term for evaluation that involves direct user testing). For example, in one intrinsic evaluation the developers might look carefully at the requirements scenarios and basic-level scenarios and ask whether the latter truly address the former. The findings of intrinsic evaluations are typically complex. New possibilities arise and the developers may end up envisioning functionality that does not merely address user requirements, but in fact improves upon them. Conversely, unanticipated obstacles arise and the developers may end up developing information scenarios that do not fully address the user needs described in the requirements scenarios. The purpose of intrinsic formative evaluation is to re-think the relations among scenarios and to identify new alternatives.

Pay-off formative evaluation combines aspects of the other two approaches. The goal for this kind of evaluation is formative, that is, to improve the design. But the method for pay-off evaluation is empirical, not analytical; it consists of testing users. Pay-off formative evaluation is used when a prototype has been created — in particular, when enough of the system has been built to allow user testing, but there is still time and resources to re-think and revise the design.

The Prototyping Cycle

On the right of the design sub-stages in Figure 2.3 is the prototyping cycle. This is an iterative sub-process; the current design is embodied as a prototype and then evaluated in order to guide redesign (formative evaluation). A prototype can take many forms. For example, a basic-level test scenario could be a prototype. Test users could read the scenario, critique it, enact it, explain it to peers, etc. More appealingly perhaps, one could film a day-in-the-life video in which an actor uses simulated technology to carry out the activities of a basic-level scenario. Test users could view this and be asked a set of specific

questions to determine whether the envisioned scenario meets their requirements and how it might be detailed in terms of information and interaction scenarios.

Prototypes often allow some degree of interaction. The simplest example is a set of paper or cardboard screen mock-ups; actions taken by the test user determine which mock-up is displayed next. A similar approach can be taken with user interface authoring tools that store and sequence a set of display designs.

Of course the most conventional notion of a prototype, and the most powerful, is that of a software system that partially implements the system being designed. Users can carry out the limited set of tasks the prototype supports, and their successes, difficulties, and feelings about the prototype can be measured, collated, and summarized to suggest possible design changes. As the prototype covers more of the target functionality, developers can study a greater variety of user tasks, and can allow test users to carry them out in a more open-ended and realistic manner.

The prototyping cycle is a continuing flow from design, to prototyping, to formative pay-off evaluation, back to design. In our model, prototyping starts early with low-fidelity (e.g., paper and cardboard) prototypes, and continues as the system grows. While we agree with Brooks (1975) that developers should be prepared to throw prototypes away and start again, we do not see prototyping as necessarily a separate process only for clarifying requirements within the design process. In some situations a prototype can be successively refined to become the delivered system.

The Documentation Cycle

To the left of the design sub-stages in the figure is the documentation cycle. This cycle documents the evaluation activities and the current state of the design, as project rationale. The project documentation records why and how various issues were addressed in the design. It lists open questions and problems that were identified in the formative evaluation. Over time, the project rationale becomes both an analysis of the current design, and a record of earlier stages in the design process. This kind of documentation helps to avoid thrashing, that is, revisiting the same design issues, only to try and finally reject the same approaches. It provides a memory or history resource to the development team.

Project documentation can also help developers consolidate lessons learned in a software development project, so that they are better able to apply their own experience in subsequent projects. To the extent that developers learn from each project, they will become more competent professionals. For this purpose, project documentation can become a sort of “research literature” for software development, providing an empirical foundation for design guidelines. Developers solve many problems, and some of their solutions are both technically significant and general; documenting these results can benefit both the individuals and the profession.

The project documentation and the current state of the design are combined to develop the user documentation for the system (reference materials, instructional manuals,

online help, and so forth). Users can benefit from understanding why the system was designed as it is, as well as how to use it. Indeed, conveying the rationale for a design to users can reassure them that the design decisions were not arbitrary and thereby motivate them to learn the system. User documentation is also itself an important test of a system's design: Good designs are easier to explain.

The documentation cycle is not as coherent or cyclic as the prototyping cycle. Documentation activities do feed back to design; for example, problems encountered when writing user guides help to identify problems in the user interface and underlying functionality. And it is quite common for developers to realize that the rationale for certain design decisions is weak when they record that rationale in project documentation. Thus, documentation activities can trigger and guide redesign. But because the impacts of the documentation stages are less significant and pervasive than those of the prototyping cycle, we show the relationships as one-way arrows.

Other Approaches to Usability Engineering

All software development methods — and this includes all usability engineering methods — struggle with the tension between a linear waterfall of sub-processes with clearly specified dependencies, and a flexible prototyping approach. The resolution of the tradeoffs between these two perspectives will almost always incorporate a linear flow of information and development, accompanied by iterative feedback and reworking.

A good example is Deborah Mayhew's (1999) recent handbook. Mayhew's usability lifecycle incorporates five major phases: requirements analysis (in which usability goals are defined), conceptual model design, screen design, detailed user interface design, and installation. The model clearly integrates prototyping and iteration with a document-oriented waterfall. All of the phases except requirements analysis include assessment activities that support iteration within the phase. The three central phases — conceptual model design, screen design, and detailed user interface design — also have a combined assessment of whether all functionality has been addressed; if that test fails, the process returns to requirements analysis.

Mayhew's lifecycle model corresponds closely to the basic requirements-design-evaluation waterfall of SBUE. The key difference between the models is the role scenarios 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 authoring-tool mock-up. In SBUE the output of any phase includes user interaction scenarios. A secondary contrast is that SBUE places explicit emphasis on prototyping and documentation as supporting activities that take place concurrently throughout the development process.