MORGAN & CLAYPOOL PUBLISHERS

Conceptual Models

Core to Good Design

Jeff Johnson
Austin Henderson

SYNTHESIS LECTURES ON HUMAN-CENTERED INFORMATICS

John M. Carroll, *Series Editor*

Conceptual Models

Core to Good Design

Synthesis Lectures on Human-Centered Informatics

Conceptual Models: Core to Good Design

Jeff Johnson and Austin Henderson

www.morganclaypool.com

ISBN: 9781608457496 paperback
ISBN: 9781608457502 ebook

DOI 10.2200/S00391ED1V01Y201111HCI012

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON HUMAN-CENTERED INFORMATICS

Lecture #12
Series Editor: John M. Carroll, *Penn State University*
Series ISSN
Synthesis Lectures on Human-Centered Informatics
Print 1946-7680 Electronic 1946-7699

Conceptual Models

Core to Good Design

Jeff Johnson
UI Wizards, Inc.

Austin Henderson
Rivendel Consulting & Design

SYNTHESIS LECTURES ON HUMAN-CENTERED INFORMATICS #12

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

People make use of software applications in their activities, applying them as tools in carrying out tasks. That this use should be good for people – easy, effective, efficient, and enjoyable – is a principal goal of design. In this book, we present the notion of Conceptual Models, and argue that Conceptual Models are core to achieving good design. From years of helping companies create software applications, we have come to believe that building applications without Conceptual Models is just asking for designs that will be confusing and difficult to learn, remember, and use.

We show how Conceptual Models are the central link between the elements involved in application use: people's tasks (task domains), the use of tools to perform the tasks, the conceptual structure of those tools, the presentation of the conceptual model (i.e., the user interface), the language used to describe it, its implementation, and the learning that people must do to use the application. We further show that putting a Conceptual Model at the center of the design and development process can pay rich dividends: designs that are simpler and mesh better with users' tasks, avoidance of unnecessary features, easier documentation, faster development, improved customer uptake, and decreased need for training and customer support.

KEYWORDS

conceptual model, conceptual design, concepts, task-domain, tasks, tools, task-to-tool mapping, object/operations analysis, interaction design, user interface design, application design, software design, design, usability, software development method, software development process

Contents

Preface

We have been interested in Conceptual Models for years. We both lived through the rough and tumble days of inventing the future at Xerox and understand just how hard it has been for the world to develop applications that work as well as they do. A continuing subject of discussion for all concerned has been the "model of the system," the view of the application that the designers hope people will adopt when using the application.

Yet in all this time, there has been a lack of clarity about exactly what these models are. This is not entirely surprising, seeing as there are so many different kinds of models, and modeling is such an endemic effort in the development of systems.

For that reason, in 2002 we wrote an article for *interactions* magazine – a 2500 word attempt to encourage designers to "begin by designing what to design". In the intervening decade, we have received sporadic reports that that article has been helpful to some designers and developers. However, conceptual models do not seem to have become common in accepted practice.

We also noticed that our own interest in conceptual models continued to evolve. We realized that there was much more to say than we said in the Interactions article. We felt that conceptual models should be discussed more thoroughly and in a place that was readily available to all engaged in developing applications or studying how to develop them.

This book is the result.

Jeff Johnson and Austin Henderson
November 2011

Acknowledgments

We thank Jack Carroll for offering to publish this in his "Human-Centered Informatics" series at Morgan & Claypool. His encouragement has been supportive. We thank Diane Cerra for her help in setting direction and managing the mechanics of this undertaking. We thank our colleagues Robin Jeffries, Jon Meads, Susan Fowler, and Nigel Bevan for their helpful feedback on our first draft. We also thank Jon Meads for allowing us to use two of his diagrams (Fig. 8.3 courtesy of Jon Meads, copyright © 2011 Jon Meads). We thank those who helped edit and design this book.

Finally, we each have people we would like to acknowledge.

Austin Henderson: I want to acknowledge many people for conversations over the years that have touched on conceptual models, particularly, Tom Moran, Stuart Card, John Rheinfrank, Shelley Evenson, Don Norman, David Asano, Hugh Dubberly, Jed Harris and, of course, most centrally, Jeff Johnson. I want to thank my wife Lynne for invaluable support and encouragement while this work has been underway.

Jeff Johnson: I acknowledge the many insights I have gained as a result of discussions — and arguments — about interaction design and conceptual models over the years with many colleagues with whom I have worked: Robin Jeffries, Bonnie Nardi, Steve Whittaker, Terry Roberts, Chuck Clanton, Stuart Card, and of course, Austin Henderson. I also acknowledge the support and patience of my wife Karen Ande as this book was being written.

Jeff Johnson and Austin Henderson
November 2011

Introduction

This book presents and argues that a good Conceptual Model (CM) should be at the core of the design of every artifact that people use to help them get their work done. This includes software products, electronic appliances, and web services, but also products more generally and even human services[1].

Those unfamiliar with interaction design often consider it to be designing the user interface or "skin" of an application: the controls, displays, error messages, use of colors, layout, etc. However, the interaction design for an application is more than just the controls, displays, etc. that comprise its user interface. The interaction design goes deeper than the skin: it includes all concepts that the application's user interface exposes to users, and it includes the sequences of operations that users execute to accomplish the application's supported tasks. The interaction design has roots that extend deep into an application's *architecture* and implementation — including into any back-end services the application uses. Thus, an interaction design consists of *concepts, task-flow,* and *presentation*.

A key part of interaction design is creating a *conceptual model* of an application. The purpose of conceptual design — of creating a conceptual model — is to get the concepts and their relationships right, to enable the desired task-flow. Obviously, it makes sense to get the concepts and their relationships right before designing how those concepts will be implemented or presented. In other words, start by designing *how the user would ideally think about the application* and its use in supporting tasks. Indeed, shape the whole development process around creating and delivering a good conceptual model.

A FEW EXAMPLES

Let's consider examples of conceptual models. Here are a number of possible pairs of alternative conceptual models, any of which could be quite acceptable given different circumstances. However, the designer must choose one of them (or invent yet another alternative) because these alternative conceptual models are incompatible.

Assume you are designing the following:

- Support for discussion grouped around topics. Is the structure:

  a) a set of topics, each with a flat list of responses, or

  b) a hierarchy of postings, each potentially with responses?

[1] For simplicity, this book uses the term "application" to cover all such technologies.

- An application for creating newsletters. Is a newsletter:
    a) a list of items, or
    b) a set of pages, each with layout of items?
- A calendar for managing events and appointments. Is the calendar:
    a) a list of days, or
    b) a list of events?
- An application for creating and editing organization charts. Is an organization chart:
    a) a tree structure of organizations and employees, and attributes thereof, or
    b) a set of organizations and employees, linked together in a variety of ways?

THE BENEFITS OF CONCEPTUAL MODELS

This book argues that a conceptual model:

- helps focus the design of the application by coupling the design to the tasks that the user is doing;
- supports having a good process for developing that design into a product; and
- makes using the application easier for the user.

Since the design, process, and experience of use are all informed by the conceptual model, these all feed off each other and grow together.

ORGANIZATION

The book is organized as follows. (Please see the figure on page 3.)

Chapters 1 and 2 set the context within which conceptual models are important. Chapter 1 (Using Tools) reviews the role of tools in helping people to get work done. It introduces key concepts and terms (e.g., task domain, task, application, mental model, conceptual model, user interface, implementation). With the place of tools established, Chapter 2 (Start with the Conceptual Model) provides sketches of several alternative ways that people carry out design, starting with the task, the user interface, or the implementation; instead it is argued that the place to start is by designing the conceptual model.

These initial two chapters are intended to provide those new to the design of tools with sufficient background knowledge to understand the rest of the book. Those experienced in the theory and/or practice of designing tools may want to skip Chapters 1 and 2 and start with Chapter 3. However, first they may want to check how our terms align with those with which they are familiar.

Chapters 1-2. Context of CMs: Designing tools to help people get tasks done:

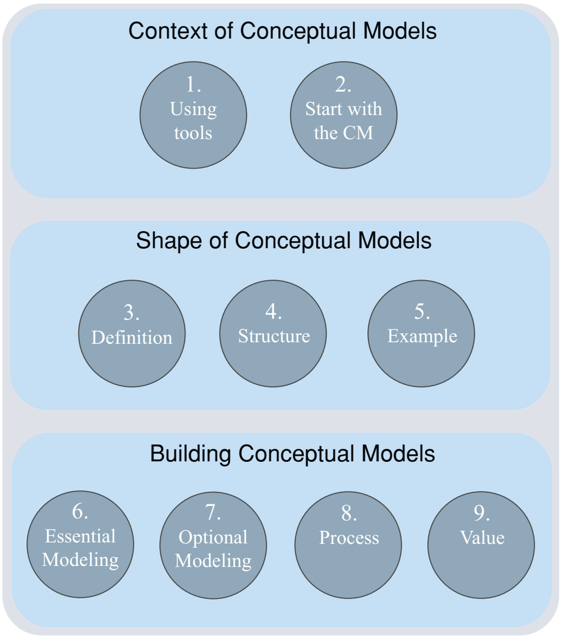
1. Using tools to do work;
2. Where to start design.

Chapters 3-5. Shape of CMs: Introduction to conceptual models:

3. What they are and are not;
4. Their parts and structure;
5. A full example.

Chapters 6-9. Building CMs: Applying conceptual models to the practical realities of design:

6. Essential modeling issues;
7. Optional modeling issues;
8. Recommended processes;
9. Value of CMs.



Chapters 3 – 5 explain what conceptual models are. Chapter 3 (Definition) introduces conceptual models and the purpose they serve. Chapter 4 (Structure) describes how they are structured (objects/attributes/operations) and some common ways of denoting them (e.g., UML, concept maps). Both Chapters 3 and 4 are filled with parts of examples. Chapter 5 (Example) provides a much more complete, and so larger, worked example. Most readers will want to focus on Chapter 4, with Chapters 3 and 5 providing scaffolding.

Chapters 6 – 9 discuss building and using conceptual models. Chapter 6 (Essential Modeling) describes how common configurations of concepts (e.g., types, specialization) can be expressed using the objects/attributes/operations structure of conceptual models. Chapter 7 (Optional Modeling) raises some advanced issues that designers may choose to address in their conceptual models. Chapter 8 (Process) discusses how the conceptual model can and should play a continuing role in enabling the many perspectives on a design to create and maintain alignment with each other throughout the course of the development of tools. Chapter 9 (Value) discusses the benefits that conceptual models can bring to the development of tools, and therefore to users in their work of getting their tasks done.

CHAPTER 1
Using Tools

Before exploring the role and value of conceptual models (CMs) in tool development and use, it is helpful to have a larger context within which to understand the use of tools. This chapter describes a framework, illustrated in Figure 1.1, for the use of tools, as a context for the rest of the book.

1.1 ACTIVITY

In their work and their play, people are engaged in *activities*. For example:

- They manage their photographs, their finances, their time.
- They send e-mails, write books, compose music, create personal history albums.
- They shop, converse, attend concerts, travel.

These activities are sometimes simple, but more often they are rich and complex. They evolve as time passes. They can overlap each other, with one activity serving a number of purposes. They sometimes finish; they often go on indefinitely. Sometimes we describe them; often we don't.

1.2 TASKS

As part of their activity, people carry out *tasks*. A task is a well-defined chunk of activity that has a goal that can be achieved with actions, some of which can be delegated to tools. For example:

- Jack makes a deposit to his bank account.
- Jill buys the book *Harry Potter and the Sorcerer's Stone* for Tim.
- Hillary tweets about the conference.

1.3 TASK DOMAINS

All the tasks associated with an activity, considered together, are referred to as a *task domain*. For example, the task domain of financial transactions could include making deposits, reviewing balances, making withdrawals. A broader (more encompassing) task domain might include all of these, together with bill paying, transferring money between accounts, and budgeting.

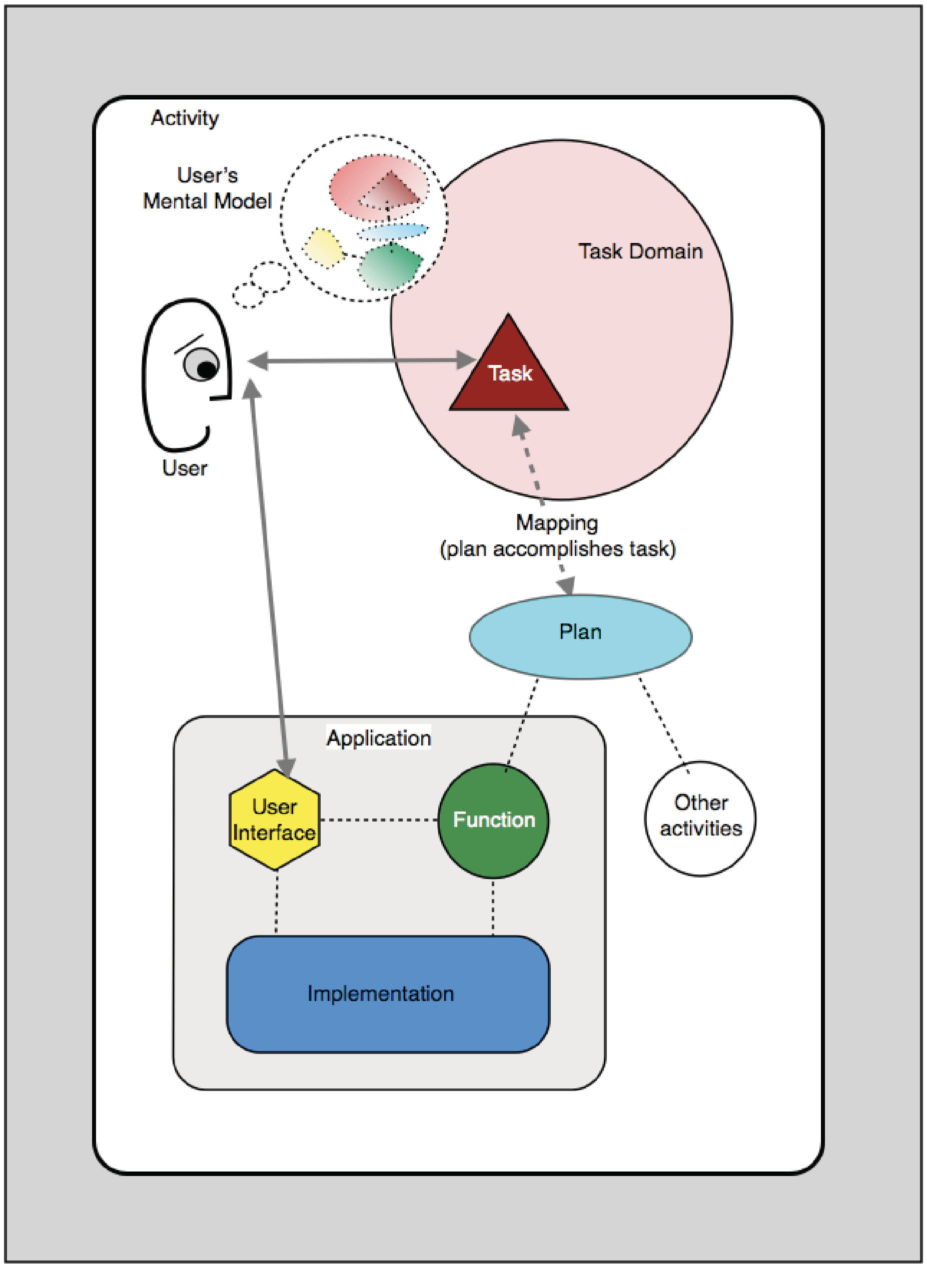


Figure 1.1: Using a tool (an application).

1.4 TOOLS

From the perspective of people with things to do, technology is of no interest in and of itself. What matters is *technology in use*, that is, *tools*. A tool is technology that people use to help them carry out tasks in a task domain. For example:

- Jack makes his deposit using an ATM at the bank.
- Jill uses her iPhone to remind her which book Tim wanted, and her credit card to pay for it.
- Hillary tweets about the conference using Twitter on her Blackberry.

1.5 FUNCTION

An application is used as a tool to help with a user's task because it adds value to their activity. For example, a photocopier makes copies: it takes an "original" (a document) and produces "copies" (more documents) with the same content, but possibly modified in some way (e.g., reduced in size, stapled).

This valuable service — what the application does and the user employs in getting their work done — is the *function* of the application. The *function* of a photocopier is that it makes copies.

1.6 TASK-TO-TOOL MAPPING

Tasks may be accomplished with different tools. For example, a person can sell stock through a website, an automatic phone tree, or a broker. So at some point a person has to find out what tools are available, and learn how each can be useful for the tasks that the person wants to carry out as part of their activities. The person has to make the connection between task and tools, a *plan* for using the functions of the tools to accomplish the task. Tom Moran has called it the External-Task/Internal-Tasks (ETIT) mapping [Moran, T.P., 1983].

Effective tools can have strong influence on how people conceptualize their tasks, both because the tasks can be mapped into the tools, and because the capabilities of the tools can extend how people see their tasks. For example:

- climbing a tree is easier if you wear tree-climbing spurs, but if you have a ladder, the need to climb a tree (e.g., to pick its fruit) disappears entirely;
- the availability of spreadsheet programs not only leads people to recast computations as spreadsheets, but also to explore alternatives, something that is much easier with a spreadsheet than a calculator.

Tools, thus can also become "tools for thought". Richard Young in "The Machine within the Machine" has talked about how tools reshape not only work, but how people think about work [Young, R.M., 1981].

1.7 DESCRIBING TOOLS

This raises the central question that motivates this book: How are tools thought of by people? What is the information that a person needs concerning what a tool does, so that they can then figure out whether that tool will help them with their task? Correspondingly, how is that information expressed: how are tools described?

1.7.1 TASK-BASED DESCRIPTIONS OF TOOLS

A common way to describe tools is to describe a few representative tasks, and then describe how to interact with the tool to accomplish those tasks. This description incorporates the mapping from task to tool. Such a description can be helpful if the task at hand is one of those supported. For example, bank customers usually open safety deposit boxes inside their bank, so an ATM cannot help with that task.

However, describing how to perform a few tasks may not make clear what task domain the tool addresses, nor make it easy for the user to infer how the tool behaves over all the other tasks. For example, instructions for a new BlueRay DVD player with an ethernet connection describes a number of common tasks including how to connect the ethernet, but does not make clear what tasks can be performed using the ethernet connection.

Sometimes task-based descriptions of tools are given in terms of operating its user interface. For example, instructions for numeric calculators give examples of how to carry out a few representative calculations (to add 34 and 56, push c,3,4,+,5,6,=), and assume that users can see the pattern and generalize it to all calculations. The difficulty with that is that even when a person completes a task successfully, there is no guarantee they will understand how the actions they have performed have achieved the effect they have achieved. For example, explaining how to put slide numbers onto presentation slides in terms of the menus to use and the checkboxes to check does not provide any understanding of how to get those slide numbers in any other form or position, and indeed whether that is even possible.

1.7.2 MODEL-BASED DESCRIPTIONS

One of the best ways to describe a tool is to provide a model of the sorts of things that the tool can manipulate, and the sorts of manipulations that it can perform. Such models provide concepts (objects and operations) that a person should be aware of in order to make use of the tool.

For example, a tool for banking might be described as manipulating bank accounts, enabling transactions (including deposits, transfers, and withdrawals), providing reports. Note that there is no mention of how these operations are achieved, but there is adequate information for deciding, for instance, that this tool will indeed help you make a deposit and that it will not help you wire funds to a bank in Europe.

The advantage of models as descriptions of tools is that they provide enough information to help a person make a task-to-tool connection, without overwhelming them with the details of either how to interact with the tool or how the tool achieves the task. This way of describing tools is the main focus of this book.

1.8 SEEING THE MODEL THROUGH THE USER INTERFACE

While people can *think* about using a tool at the functional level, to actually *do something* using the tool they must access its functionality through some interactional mechanisms — through a user interface.

For example, the user interface for selling stock through a website involves information displayed in columns and lists, areas for entering text, links and clicking, pull-down menus, and much, much more. The same tool may also be accessible through a different user interface based on a telephone, where the interaction involves multi-level menus (phone trees), listening to choices, selecting choices by typing numbers on the phone, entering information by typing numbers, confirming by typing numbers, and so on.

A user interface delivers an application's function to a user. A clear conceptual model will focus the user interface to stay "on message," delivering all of that model and only that model.

1.9 IMPLEMENTATION ARCHITECTURE AND IMPLEMENTATIONS

These days, many tools are realized as software applications that run on computer hardware. Developing such software requires additional descriptions of the application, such as structural designs and implementation architecture. For example, the implementation architecture for a financial application might include components for security, databases, networks, and servers. User interfaces also require implementation architecture. For example, they can include components such as browsers, windows, menus, and screen layouts.

Implementation architecture is used to shape the software itself, both the functionality and the user interface of the design. Taken together, the implemented functionality and the implemented user interface constitute the application.

1.10 MENTAL MODELS

To use an application, a user must have some understanding of it. Such an understanding is called a "mental model" because it is in the user's mind. Regarding both the application's function and its user interface, the user's mental model is partial, often even incorrect.

A user will also have an understanding of the task that they are trying to accomplish. This too may be partial and/or incorrect.

Every user has their own mental models of their tasks and tools. For example, although different users are employing the same copier, they may understand it differently. They may understand some aspects well, and others less well or not at all. Also, a user's mental model may well change as a result of experience with the application.

Therefore, mental models are personal, partial, uncertain and dynamic.

1.11 CONCEPTUAL MODELS

To develop applications, developers (including all members of the team) need their own understandings of the task and tool. These understandings are more stable, because they are abstracted over the activity (real and imagined) of all intended users. Figure 1.2 builds on the diagram presented earlier (in Fig. 1.1) by adding components representing the designers, their intended model of the application, and the relationship of those components to the application's functionality and user's mental model.

While users will focus on one task at a time, the developers must design for the whole task domain and all foreseen user-tasks (use-cases) within it. Before the application is complete, the developers will have to address all aspects of the application. However, a central part of the design will focus on the functionality needed by the user in carrying out all the tasks from the task domain.

This designers' model of an application is called the "conceptual model" because it describes the concepts that a person using the tool will be able to manipulate. Chapters 3 – 8 explore the structure and use of such models.

Conceptual models may be thought of as the *ideal mental model of the function* of the application. As the rest of this book argues, they are key to good design.

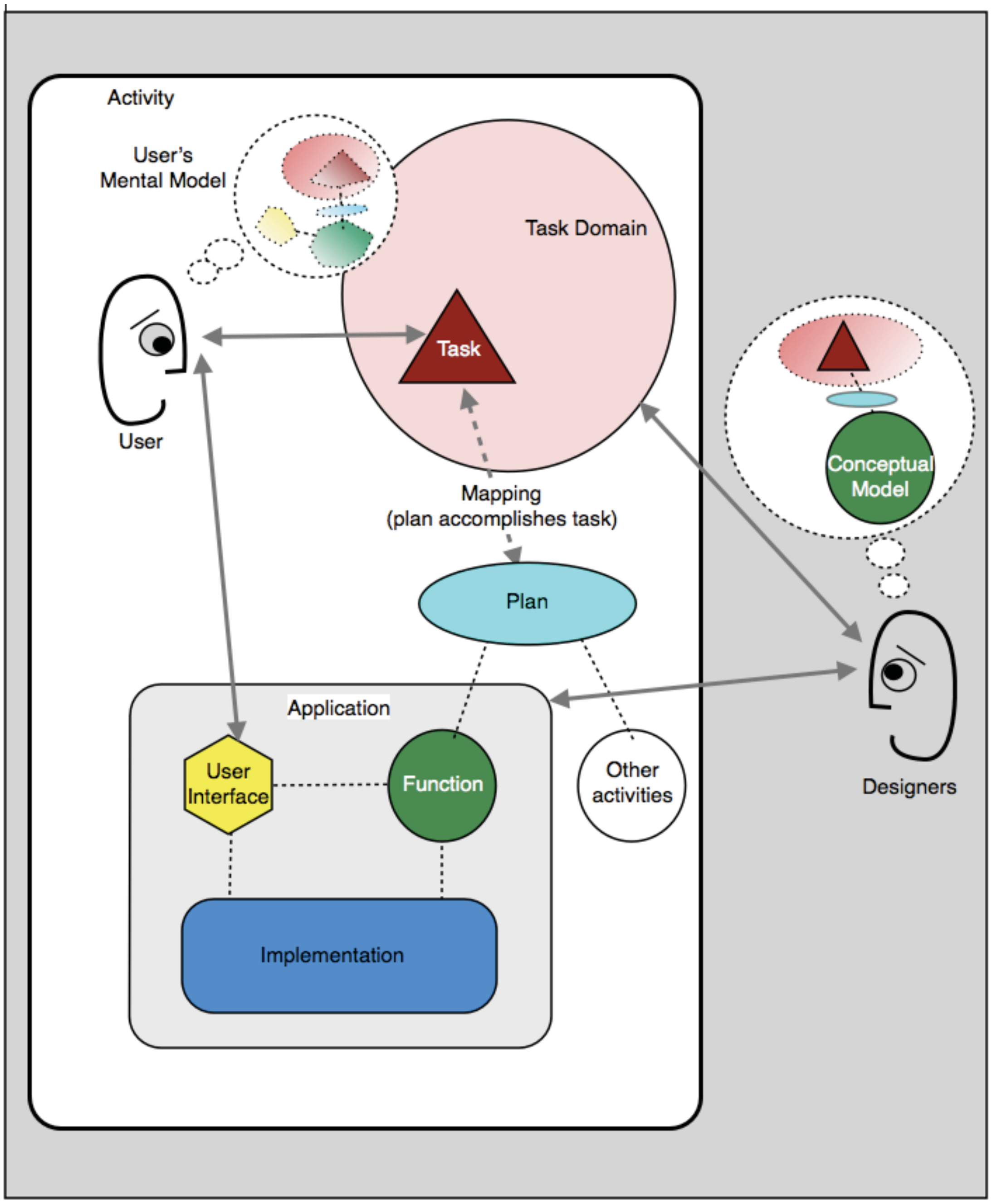


Figure 1.2: Designers' model of a user using an application.

CHAPTER 2

Start with the Conceptual Model

Suppose you are designing an application. You've learned why your organization is creating it and know who the targeted customers and users are. You've spent time with some of those people to understand their activity and how the application will fit into it, and gathered functional requirements from them and from Marketing. In short, you've performed a business-case analysis, a market analysis, a task-analysis, created user profiles, and are ready to start designing the application. What's your first design step?

2.1 DESIGN (AND IMPLEMENT) FUNCTIONALITY?

Some software developers start designing an application by designing the implementation of its functionality. They define the application's architecture — platform, functional modules, and communications protocols — and then immediately begin implementing the modules and protocols on the chosen platform. Only after the functionality is complete or nearly complete do they begin thinking about how people will operate the application.

This was the prevalent practice in the early days of software development, when engineers and scientists were still the main users of software applications. The design attitude was: "Get the functionality right, then slap a UI onto it. The users can adapt."

That approach never worked very well, even when the application users were highly trained scientists and engineers, but it became completely untenable when the use of computers and computer-based products and services began to spread to the general population in the mid-1980s. In the 1990s, the "functionality first, then UI" approach was largely supplanted by the approach described in the next section.

However, the old "functionality first, then UI" underwent a resurgence in the early 2000s, as Agile software development practices became popular — a resurgence that continues to this day. Chapter 8 explains that this is based on an overly-simplistic but widespread view of Agile development, and that Agile development does not necessarily mean focusing on implementation before user interface. However, for now suffice it to say that a significant number of software development teams today jump right into coding the functionality of an application before they figure out how people will operate it.

Fortunately, many software designers and developers know that the "functionality first, then UI" approach usually yields software applications that are hard to learn, hard to operate, and have questionable utility. Therefore, they reject that approach and instead begin designing an application by focusing on how people will use it.

2.2 DUPLICATE THE EXISTING ACTIVITY?

The obvious place to turn to see how people will use new technology is to look at what they are doing now. By spending time with people working in the target task domain, their activities can be observed and the tasks they do identified and understood.

However, understanding how people currently work does not mean that designers should simply design software to support the work as it currently is. Such an approach misses the fact that the tools currently being used are shaping how the activity is being done, just as the activity affects the tools that are used. In other words, the activity and the tools used to do it are mutually co-defining.

For example, if we observe how someone commutes to work, we would see quite different behavior depending on whether the person walks, rides a bicycle, drives, or takes public transit. The tools a person chooses — transportation methods in this example — strongly impact how the person executes the task.

Designers of new task-support technology should take care not to simply duplicate the details — including flaws — of the tools people currently use to perform the task. For example, simply replacing a horse with an engine yields not a car but rather a horseless carriage, complete with reins and wagon wheels, with the drivers sitting in the open high above the ground. Similarly, studying traffic congestion in a simplistic way might suggest where to add road lanes, but probably wouldn't suggest building a light-rail transit system.

Instead of simply designing new tools to support the old-tool-dependent details of how people currently do a task, designers should design new technology to support how people *want to think about* doing their tasks.

2.3 DESIGN (SKETCH) THE USER INTERFACE?

Unfortunately, many designers, especially those new to application design, believe that "focusing on how people should think about doing the task" means starting by sketching the control panels and dialog boxes of their application. Such initial sketches are usually high-level and low-fidelity, showing only gross layout and organization.

However, starting by sketching the user interface amounts to starting with how the application *presents itself* to users. This tends to lead designers into addressing problems about the presentation, and divert them from the problems of designing what the presentations are supposed to present. At worst this can result in a usable interface to useless functionality. At best it tangles the two issues together, and can lead to functions that only be delivered through a single user interface. For example, personal-finance applications could be designed around records that look like checkbooks, making them hard to extend for use in cars where access would be through voice.

2.4 START WITH *CONCEPTUAL* DESIGN

It is better to start by designing how people think about their tasks: the *concepts* users will think about when using the application, and how those concepts fit together into a structure. That is, it is best to begin the design phase by designing a *conceptual model*. Designing a conceptual model is called *conceptual design*.

Designing a coherent, task-focused, conceptual model is the most important step in designing a user interface for an application. It has been called the "jewel in the crown" of user-interface design steps – the step with the largest impact on whether the application will make sense to its users [Card, S., 1993, 1996]. Unfortunately, conceptual design is often skipped in software development. The result is a marketplace – and a Web – full of incoherent, overly-complex applications that expose irrelevant concepts to their users.

2.5 IMPORTANT DECISIONS

Conceptual design decisions matter. Depending on how designers conceptualize an application, users of the application will think differently about using it: the objects will be different, the operations users can do on them will be different, and how users work will be different. Confronted with different ways of thinking (for example the pairs presented in the Introduction), designers could try to avoid choosing, to have it both ways, but then users will get a confused understanding of the application and confused direction on how to think about using it. *Not choosing* is tempting, because these decisions are difficult to make: they usually require tradeoffs between simplicity and power and they always require understanding users' tasks. But whether or not designers create a conceptual model, in the end the application will have one, even if it is just an unintentional result (often incoherent and unclear) of the rest of the design process.

Conceptual design decisions are often tough, but they are essential. They are better done right up front, before they are encumbered or locked-in by downstream user interface design and implementation decisions. Get the bone structure right first, then flesh it out.

By carefully crafting an explicit conceptual model focused squarely on the target task-domain, and then designing a user interface from *that*, the resulting product or service will have a much better chance of being as simple as the task-domain allows [Norman, D.A., 2010], coherent, and easy to learn. In contrast, designers who jump straight into laying out screens and dialog boxes, or choosing controls and widgets from a user-interface toolkit, are much more likely to develop a product or service that seems arbitrary, incoherent, overly complicated, and heavily laden with computer-isms.

Designers with strong backgrounds in human-computer interaction (HCI) and user-interface design are probably well aware of the value of conceptual models. However, our experience with our clients indicates that conceptual models of this sort are almost completely unknown outside of the HCI community, especially among web-designers and software programmers.

2.6 CONCEPTUAL DESIGN'S PLACE IN SOFTWARE DEVELOPMENT

A good conceptual model is central to a good product. A good development process will therefore keep the design of its conceptual model clearly in focus. This book therefore argues that conceptual models can be central not only to the design of good products, but also to the processes of developing them.

Chapter 8 (Process) discusses in detail where and how conceptual models fit into the software development process, but for now it suffices to say that conceptual design is at the center of development: it focuses the design process, it coordinates the design activities, it frames the implementation design, and the design of documentation and support. Most design decisions engage it. It is at the boundary between different perspectives, and a bridge between them.

CHAPTER 3

Definition

Before we explain how to design conceptual models, we must make clear what they are (and are not).

3.1 WHAT A CONCEPTUAL MODEL IS

3.1.1 HIGH-LEVEL DESCRIPTION OF AN APPLICATION

A *conceptual model* is a high-level description of an application. It enumerates all concepts in the application that users can encounter, describes how those concepts relate to each other, and explains how those concepts fit into tasks that users perform with the application. In so doing, the conceptual model describes abstractly — in terms of user-tasks, *not* keystrokes, mouse-actions, or screen graphics — what people can do with the application and what concepts they need to understand in order to use it.

More specifically, a conceptual model specifies and describes:

- the target task-domain, purpose, and high-level functionality of the application;
- the ***concepts*** the application exposes to users, including the task-domain data-objects that users create and manipulate, their user-visible names, their attributes (options and settings), and the operations that can be performed on them;
- the ***relationships*** between these concepts;
- the ***mapping*** of task-domain concepts to application concepts, usually expressed as task-scenarios written in terms of the conceptual model.

As a rough and simple example, consider an alarm clock. A conceptual model for it might state the following.

- The clock stores the current time of day, continually updating it to track the passage of time.
- It displays the current time constantly.
- Users can set the current time.
- Users can set an alarm at a specified time, or no alarm.

- When an alarm is set and the current time equals the set alarm time, the alarm is triggered.

Our example of a simple alarm clock allows only one alarm to be set at a time. It does not include concepts of *multiple alarms*, possibly with different *alarm-indicators* (e.g., various sounds, flashing lights, radio ON, start music-player) for each alarm. It *could* include such concepts if the manufacturer wanted to offer a fancy alarm clock and could justify the extra complexity and cost. However, if support for multiple alarms is not a *functional requirement*, such concepts would be better left out of the conceptual model.

Similarly, the manufacturer might want the clock to indicate the *current date* and *day of the week* as well as the *time of day*. If so, the conceptual model must include those concepts. However, features should not be added lightly because they often increase the number of (and interactions between) concepts in the conceptual model and hence its complexity, and hence the complexity of the product.

The alarm clock could be made even simpler. Like mobile phones, it could obtain the time from an external source, such as a cellular link, GPS, or wi-fi Internet. If the external connection and querying of the time were totally automatic and error-free, the conceptual model for such an alarm clock could be very simple indeed:

- It displays the current time constantly, never needing to be set.
- Users can set an alarm at a specified time, or no alarm.
- When an alarm is set and the current time equals the set alarm time, the alarm is triggered.

What if getting the time from an external source were *not* fully automatic and error-free? Any exposure of that mechanism to users would make the clock immensely *more* complicated than an ordinary alarm clock. In addition, an alarm clock that got the time from external sources would cost more than an ordinary clock because it would need more electronic components. Sidebar 1 provides an example of a more complex application.

3.1.2 BASIS FOR USERS' UNDERSTANDING OF THE APPLICATION

A conceptual model describes how designers want users to think about the application. In using an interactive product or service, reading its documentation, and talking with other people who use it, people develop an understanding of how the product works. This allows them to predict its behavior and generalize what they learn to new situations. Developing this understanding is an important part of learning to use a software application.

Sidebar 1: A More Complex Example

For a more complex example than the clock, consider an online catalog for a municipal library. The conceptual model might include:

- High-level functionality and purpose: e.g., the application is intended to support finding items in the library, replacing a physical card-catalog;
- Concepts: e.g., item (with attributes: title, ISBN, status; with operations: check-out, check-in, reserve), subtypes of item (e.g., book, periodical issue, LP, video), user account (with attributes: name, items checked out), librarian (with attributes: name, desk location, phone);
- Relationships: e.g., a book is one *type* of item, periodical volumes *contain* issues;
- Mappings: e.g., each item in the system corresponds to a physical item in the library.

Ideally, users' understanding of the application should match what the designers intended; otherwise users will often be baffled by what it is doing. If the designers explicitly design a conceptual model and then base the detailed user interface design on that, chances are greater that users' and designers' understanding of the product will match.

3.1.3 CLOSE RELATIVE: INFORMATION ARCHITECTURE

Web designers often include a stage in development that is similar to conceptual design: information architecture (IA). Designing an information architecture for a website involves deciding how to organize the information it will present. This yields site *structure* and content *categorization*[2]. The goal of information architecture is:

- to allow website users to find what they are seeking, and
- to promote the goals of the website owner.

Information architecture is similar to conceptual design in that it is concerned with conceptual *organization* and *structure*, rather than with presentation. Information architecture also is similar to conceptual design in that it normally concerns only *user-facing* concepts[3].

[2]Sometimes called "ontologies."
[3]Some people include implementation concerns in IA, e.g., whether an underlying database is relational, and if so how it is represented in tables.

In transactional websites and web-based applications, the conceptual model includes the information architecture, but goes beyond it to include the objects and operations comprising the site's functionality.

3.1.4 SUMMARY: WHAT A CONCEPTUAL MODEL IS

Conceptual models are best thought of as *design-tools* — a way for designers to straighten out and simplify the design and match it to the users' task-domain, thereby making it clearer to users how *they* should think about the application. The designers' responsibility is to devise a conceptual model that seems natural to users based on the users' familiarity with the task domain. If designers do their job well, the conceptual model will be the *basis* for users' mental models of the application (see "Not the Mental Model", below).

A conceptual model of an interactive application is, in summary:

- the *structure* of the application: the objects and their operations, attributes, and relationships;
- an *idealized view* of the how the application works – the model designers hope users will internalize;
- the *mechanism* by which users accomplish the tasks the application is intended to support.

3.2 WHAT A CONCEPTUAL MODEL IS *NOT*

To further clarify what conceptual models *are*, lets consider some things that they are *not*.

3.2.1 NOT TASK-LEVEL SCENARIOS OR USE CASES

The conceptual models are *not task-level scenarios* and they are not *use-cases*. In the user interface design field, *task-level scenarios* are brief stories about the domain tasks that users will have to carry out in their work. Software engineering experts define a *use-case* as: "a description of steps or actions between a user (or "actor") and a software system which leads the user towards something useful" [Bittner and Spence, 2003]. Both task-level scenarios and use-cases are expressed in a design-neutral way, so as not to specify the user interface. For present purposes, task-level scenarios and use-cases are similar enough that we will lump them together under the term "task-level scenario".

A task-level scenario for a hotel-room telephone might be: room-guest returns to room and checks to see if anyone has left voice-mail. Similarly, a task-level scenario for an online banking application might be: customer transfers funds from savings account to checking account.

Task-level scenarios emerge from study and analysis of the task domain — through interviews, ethnographies, focus groups, contextual inquiry, and other methods. They can either be *input* to the design of the conceptual model or they can *emerge* from it. Therefore, they are often included in documents about conceptual models. However, a set of task-level scenarios is not a conceptual model: task-level scenarios focus on *tasks*; the conceptual model focuses on the organization of the *application*.

3.2.2 NOT USERS' MENTAL MODEL

The conceptual model is *not the users' mental model* of the application. As mentioned in Chapter 1, users of an application form mental models of it to allow them to predict its behavior. A mental model is the user's high-level understanding of how the application works; it allows the user to predict what the application will do in response to various user-actions. Ideally, a user's mental model of an application should be *similar* to the designers' conceptual model, but in practice the two models may differ significantly. Even if a user's mental model is the *same* as the designer's conceptual model, they are distinct models.

Users' mental models can include all sorts of extraneous concepts and relationships. For example, a person's mental model of their watch might include the fact that it tends to gain time and so must be reset every week or so. It might also include the knowledge that the alarm is not loud enough to wake them, and that the watch is a family heirloom that should under no circumstances be lost or damaged.

Second, users' mental models, especially of complex software-based applications, are difficult for designers to pin down. Mental models cannot be observed directly and users can rarely describe them accurately. Designers usually must figure out users' mental models by observing people's behavior — including successes, difficulties, terminology, and questions — and asking them to explain their actions and expectations.

Finally, users' mental models are only an approximation of the designers' conceptual model. Users form mental models of an application by seeing how it behaves, reading its documentation, and talking with other users about it — in short, through their *experience* with the application. As a user's experience with an application grows, the user's mental model changes.

3.2.3 NOT A DESIGN METAPHOR

Conceptual models are not the same thing as design metaphors. Many software user interfaces are based on metaphors[4] with common objects, often from the physical world. This leverages people's (presumed) advanced knowledge of those objects. For example:

- some computer operating systems use a "desktop" metaphor [Johnson et al., 1989];
- most eReaders use a "book" metaphor;
- some electronic address books use a "Rolodex" metaphor;
- most computer calculation functions use a "pocket calculator" metaphor [Johnson, J., 1985];

[4]They are more properly called "design analogies," but the term "metaphor" has stuck, so we are stuck with it.

- many Delete functions use a “trash can” or “recycle bin” metaphor.

Designing software based on analogies with familiar physical objects has disadvantages as well as advantages, such as suggesting to users that all the laws and limits of the physical world apply to the software as well, when in fact they usually don’t [Halasz and Moran, 1982, Johnson, J., 1987].

One or more design metaphors may be *included in* a conceptual model, but a conceptual model includes much besides the design metaphor, as described above and in Chapter 4.

3.2.4 NOT THE USER INTERFACE

The conceptual model of an interactive system is *not the user interface*. It is not about how the application looks or feels. It does not mention keystrokes and mouse-actions, screen graphics and layout, commands, navigation schemes, dialog boxes, controls, data presentation, or error messages. It does not say whether the software is operated through a GUI on a personal computer or by voice-commands over a telephone. It describes only objects that people must understand to operate it and operations that they can do with it. It refers only to application concepts that are based on task-domain concepts.

For example, our clock conceptual model given above says nothing about whether the clock is set by voice or by knobs on the back, whether it has rotating hands or a numerical display, or whether the alarm indicator is a sound or a blinking light. Those are *user interface issues* that are not important until the basic structure of the design has been worked out.

Our conceptual model also says nothing about how the clock looks. Digital clocks (numbers) and analog clocks (hands) vary in their presentation. There is nothing conceptual about this difference at all: at the level of the conceptual model time is time, and tracking time and making it available, however presented, is what clocks do.

3.2.5 NOT THE IMPLEMENTATION ARCHITECTURE

A conceptual model is *not an implementation architecture*. An application’s implementation architecture contains technical objects (possibly including classes, attributes, methods, and control structures) that are required to implement the application. In contrast, a conceptual model includes only concepts required by the task domain.

Although conceptual models and implementation architectures for an application are distinct, they can be related. As discussed in Chapter 9 (Value), a conceptual model often shapes the implementation architecture: some classes, variables, and methods in the implementation architecture typically correspond to concepts (objects, attributes, and operations) in the conceptual model. For example, a BankAccount class in the implementation architecture underlies the concept of a bank account object in the conceptual model. However, an implementation architecture also includes implementation mechanisms that are of no concern to users and thus have no place in the conceptual model. It is unfortunate that the terminology of conceptual design and object-oriented software design collide (e.g., object is a term in both, with different meanings). Designers must take care when working at the intersection of these perspectives.

Both the implementation architecture and the conceptual model *describe* the application, but they describe it in different terms, for different purposes. The conceptual model describes the application in terms of concepts that are relevant to tasks to be supported by the application. The implementation architecture describes the application in terms of the components, modules, and layers of which it is constructed. Most of the technical objects in the implementation are of no concern to users, and therefore should not be exposed to them.

Our clock conceptual model, for example, says nothing about the internal workings of the clock, e.g., if it is analog or digital, if it has a quartz crystal and step-down gears, or even if it is electrical or mechanical. Those are *implementation issues* that are irrelevant to using the clock. There is nothing at all conceptual about the difference between digital and analog where clocks are concerned: users think of them as being the same — they just tell time.

Pure implementation objects, such as the *database* underlying a personnel management application, are of no interest to the application's users and so should not be exposed to them. Therefore, such concepts have *no* place in the application's conceptual model. Including them in a conceptual model detracts from the task-focus and reduces the learnability and usability of the product, no matter how much effort goes into designing the user interface. A clock, for example, might have a *time register* where the current time is stored, but that is of no interest to the clock's users, so a conceptual model would exclude that concept.

3.2.6 NOT PRODUCT DESIGNER'S "CONCEPT DESIGN"

Finally, to avoid any confusion, the terms "conceptual model" and "conceptual design" should not be confused with the terms "concept design" or "design concept" as used by many product designers. They use those terms to refer to an early brainstorming phase — supported by quick prototypes, sketches, or story-boards, in order to inspire innovative, "out-of-the-box" solutions to design problems [Buxton, B., 2007, Lombardi, V., 2008].

The purpose of concept designs is to probe possible designs for the product by letting people experience them. The probe can explore any aspect of the design that affects users. It can explore the product's conceptual design, but most often a probe concerns the product's appearance or presentation.

For example, designers of innovative handheld appliances often create foam-core "concept models" of several alternative physical designs and have people pretend to use them. Car designers often build partially-functional "concept cars" to test radically new physical designs.

We don't disagree with *concept design* in the above sense. In fact, we find it a useful technique for probing. However, as stated in Chapter 2, conflating conceptual design and presentational design can be dangerous. Therefore, we believe that presentation-focused concept models are most useful *after* designers have devised a task-focused conceptual model.

3.3 DESIGN GOALS FOR CONCEPTUAL MODELS

The goal is to keep a product's conceptual model: 1) as simple as possible, with as few concepts as are needed to provide the required functionality; and 2) as focused on the users' tasks as possible, that is, with concepts that map as directly as possible onto the concepts of the task-domain.

3.3.1 SIMPLE

A conceptual model should be as simple as possible while providing the required functionality. An important guideline for designing a conceptual model is: "Less is more."

For example, in a Web search function, do the intended users need boolean search capability, or would they use it if provided? If not — if a simple keyword search function is enough — designers shouldn't complicate the conceptual model by adding boolean search capability.

Similarly, suppose a To-Do list application needs the ability to assign a priority to a To-Do item. If user-research indicates that users want two priority levels (low and high), designers should not complicate the conceptual model by generalizing the requirement to include more priority levels (e.g., 1-10) [Beyer and Holtzblatt, 1997].

Beware: attaining simplicity is not simple! Much thinking, testing, and rethinking are often required to find a conceptual model that is simple enough but not too simple to provide the required functionality.

3.3.2 TASK-FOCUSED

Conceptual models should map as directly to the target task-domain as possible. This reduces users' difficulty in translating concepts between those of their task and those of the application.

When the operations a tool provides don't match the operations of the task-domain in which a person is working, the person must figure out how to combine the tool's operations so as to comprise or approximate the task-domain operations. "Hmm. I want to accomplish X, but this tool only lets me do A, B, and C. How can I put A, B, and C together to get X?". This imposes on the person a cognitively taxing *problem-solving* task in addition to the task they wanted to accomplish: they must form a *plan* and then execute that plan. Norman and Draper [1986] call this difficulty of translating between the tool and the task-domain "the gulf of execution".

Contrast inexpensive photocopiers intended for use in homes and small businesses with large, expensive, feature-laden copiers intended for intensive use in print-shops and large businesses. When a person uses a simple copier to do a complex copying job that the copier doesn't support as directly, the person must first devise a more complex plan of operation — a sequence of steps using operations the copier *does* support together with other operations provided by the person themselves or by other tools. For example, if the goal is to make a double-sided copy from a single-sided original document, the plan might be: a) separate the original into odd and even pages; b) copy the odd pages; c) put the copies back into the paper-hopper; d) copy the original's even pages; e) collate the original and copies to put the pages in proper sequence. Then the person must *execute* that plan carefully. The need to devise a plan to bridge between the desired goal and the tool's available operations is the gulf of execution. Using a larger, expensive copier doesn't have that gulf; they provide most of the higher-level operations that users need, directly.

As this example makes clear, there might be a very direct mapping from one task to the conceptual model of an application, but the mapping from *another* task to the same conceptual model might be more complex. The conceptual model determines how much of a task-domain can be served by the application through a relatively direct mapping, and how much will require more elaborate plans. Better conceptual models will map to large task-domains with simpler plans.

Sidebar 2 provides an example of applying the task-focus design goal in a conceptual model for a complex application.

Sidebar 2: Redesigning a Conceptual Model for Protein Interaction Analysis

A company was designing software to control an instrument for biologists to use to analyze protein-interactions. An early prototype of the software was designed based on concepts related to controlling the machine, e.g., move sample-extraction pipettes 10 cm rightward, lower pipettes 5 cm (into protein sample well), activate suction, stop suction, raise pipettes 5 cm, move pipettes 20 cm leftward, lower 3 cm (to testing plate), discharge 0.01 ml of protein sample A on to testing plate, wait 30 s, activate protein-imaging lasers, etc.

Biologists who were shown this prototype rejected it completely. A typical comment was: "I don't want to control this machine; I want to analyze protein interactions." The conceptual model was re-designed based on protein-interaction analysis concepts, e.g., define and execute a *protocol* to interact *ligand*-X (3 *concentrations*) with *analyte*-Y (5 *concentrations*).

CHAPTER 4

Structure

Having explained what conceptual models are (and are not), we now describe their main components and structure. Sidebar 3 enumerates the main sources of information upon which conceptual models are based.

4.1 PURPOSE & HIGH-LEVEL FUNCTIONALITY

A conceptual model for a software application should include a very high-level description of the application's purpose and functionality. For example, the following describes the purpose and high-level functionality of a server management console application.

The high-level purpose of the Management Console is to allow server system administration and operation personnel to:

- initially set up and configure servers as desired;
- operate servers;
- reconfigure servers during operation, e.g., to reallocate resources as needed;
- diagnose problems as they arise, and either fix them or communicate with personnel who can fix the problems.

4.2 MAJOR CONCEPTS AND VOCABULARY

A second component of most conceptual models is an annotated list of important concepts embodied by the application. If the list includes the agreed-upon user-visible names for the concepts, then it also serves as a *vocabulary*[5] for the application and its documentation (see also Chapter 9). For example, the following is a partial list of concepts (and vocabulary) embodied in a server management console:

[5] Also sometimes called a *terminology*, *nomenclature*, or *lexicon*.

- **Alarm:** A notification that an event has occurred that triggered an Alarm Rule. Alarms are persistent: they remain in effect until dismissed or the condition that triggered them is no longer true.
- **Alarm Rule:** A rule specifying criteria for triggering an Alarm. Criteria can include variables that exceed a threshold value, events that exceed a threshold frequency, etc.
- **Enterprise:** Defines a boundary beyond which resources are not shared. An enterprise often corresponds to a company, as when a server-farm hosts servers for many different customer companies and doesn't want any resources or information to cross company boundaries for legal or accounting purposes. Within a company, Enterprises may also be used to separate resources for company divisions.
- **Environment Probe:** A sensor placed in a Server that measures — and thereby permits monitoring of — a variable of possible interest to, e.g., temperature, humidity, voltage.
- **Event:** Anything that is detected by the Service Processor. Events have a *Severity* attribute that indicates the event's importance: Normal, Warning, Problem. All events are recorded in a log.
- **Field Replaceable Unit (FRU):** Hardware component that can be removed from the server at a customer site and replaced (if necessary). Not all parts of a server are FRUs; some parts can be replaced only at the factory.
- **Log:** A facility of the Management Console that records events and lists them for review by users. The Console will maintain several logs for different purposes.
- **Log Entry:** A single item in a log, documenting one Event. Each log entry has a textual description, a severity, a time, a source, and other pertinent data.
- **Role:** A specific administrative job-function for a user, with specific permissions attached to it. In one Role, a given user will be allowed to perform certain functions but not others. In another Role, the same person might have different permission.
- **Non-FRU-Part:** Components that aren't Field Replaceable Units (FRUs).
- **User:** A person who uses the Management Console (i.e., any of its functions). Most users will be web-server operators, system administrators, or hardware technicians, but some will be managers of operators or administrators.

Sidebar 3: Input to Conceptual Model: User Profiles and Task Analysis

A conceptual model for an application cannot be constructed out of thin air; it must be based on something, namely, a good understanding of the users and tasks that the application is intended to support. Therefore, before starting to design an application, development team members should *observe and interview people doing the tasks* [Beyer and Holtzblatt, 1997]. The resulting data are used to construct user profiles and a task analysis.

USER PROFILES

User profiles are brief descriptions of relevant characteristics of the intended users of the application. An application will often have more than one type of user. For example, users of a server management console might include junior system administrators, senior ones, system architects, and IT managers.

A profile is created for each type of user, indicating duties, level of education, knowledge of the task-domain (high, medium, low), experience with previous versions or competitors of the application (high, medium, low), knowledge of computers in general (high, medium, low), and any other relevant attributes. Some development teams create comprehensive user profiles that include home-life details, back-stories, and even fictional pictures and names; such profiles are called *personas* [Cooper, A., 2004].

TASK ANALYSIS

In addition to understanding the users, application designers need to understand the tasks that the application is intended to support before constructing a conceptual model. There are several ways to analyze tasks. Often it is useful to analyze tasks in more than one way.

- ***Major tasks or goals (use cases).*** This is a list of all the high-level goals that users can have in mind when they prepare to do some work. For example, in the task-domain of business accounting, use cases could include:
    - prepare quarterly statements: profit/loss and balance sheet;
    - prepare end-of-year statements: profit/loss, balance sheet, tax;
    - enter transactions;
    - view a transaction;
    - edit a transaction;
    - audit an account.

- ***Task hierarchy.*** This is an enumeration — usually in outline form — of all high-level tasks decomposed into their component sub-tasks, including alternative methods of doing the task. It includes breaking sub-tasks down into smaller sub-tasks, etc. For example, editing a transaction requires finding it. A transaction could be found by browsing for it or by searching for it. Once found, some of its attributes are edited and (implicitly or explicitly) saved.
- ***Consolidated task-sequence analysis.*** This is representation of typical sequences of tasks that people working in the task-domain perform consolidated by abstracting across the observed users [Beyer and Holtzblatt, 1997]. Each sequence begins with whatever triggers it, lists all the typical steps, and ends with the accomplishment (or abandonment) of the initial goal. For example, a accountant might notice that a certain company account doesn't balance, requiring: logging in, searching for missing or erroneous transactions, correcting any errors found, rechecking the balance, and logging off.

4.3 OBJECTS/OPERATIONS ANALYSIS

The most important component of a conceptual model is an Objects/Operation analysis: an enumeration and taxonomy of all concepts that the application exposes to its users. This includes the conceptual *objects* that users manipulate, *attributes* of those objects, the *operations* that users can perform on each of those objects, and any *relationships* between objects [Card, S., 1996, Johnson et al., 1989, Newman and Sproul, 1973]. Such a taxonomy is one component of the application's *information architecture* or *ontology*, as with content categorizations in websites.

Operations in a conceptual model may be named either from the point of view of users (e.g., *review* today's events) or from the point of view of the system (e.g., *present* today's events). Throughout this book, we use the users' point of view in naming operations, but either way is OK as long as a development team follows their chosen convention consistently. However, operations listed in the product vocabulary and shown to users in the UI and user documentations should always be named from users' point of view.

Continuing with the previous chapter's example of a conceptual model for a simple alarm clock, the objects, attributes, and operations would be something like this:

Objects: clock, alarm

Attributes: clock has a *current time of day*; alarm has *ON/OFF* and *alarm time*

Operations: clock has *view current time* and *set current time*, alarm has *turn ON/OFF* and *set alarm time*

In this simple clock's conceptual model, no relationships between conceptual objects are needed.

As stated above, an objects/operations analysis presents all concepts that an application exposes to its users. Specifically, that means the following.

4.3.1 DECLARES CONCEPTS THAT THE APPLICATION WILL EXPOSE

The objects/operations analysis should be treated as a *declaration* of the concepts that the corresponding application exposes to its users[6]. The application should expose only objects, attributes, operations, and relationships enumerated in the conceptual model. Stated as design rule:

If it isn't in the conceptual model, the application should not expose it.

In our alarm-clock example, the method by which the clock triggers the alarm at the appropriate time is of no concern to users — they care only that the clock triggers the alarm at the specified time. Therefore, the alarm-triggering method is excluded from the conceptual model, and therefore from the clock's user interface.

4.3.2 INTRODUCES NEW CONCEPTS, IF NEEDED

Computer-based products and services often provide capabilities that pre-computerized tools and methods did not provide, so concepts new to a task domain sometimes appear in the conceptual model for a new product or service. Some examples:

- Most software-based calendars provide repeating events, in contrast with paper calendars, which require people to record recurring events by writing the same event on multiple dates.
- In a physical filing drawer, each hardcopy document can be filed in only one place, whereas electronic documents in an computerized document system can be filed in multiple folders simultaneously.

[6]Users engage the application through its user interface (UI), so the UI is how the application's conceptual model is exposed.

- Modern digital audio recorders allow recordings to be played back at different speeds either just like an old-fashioned tape recorder — altering the pitch of the recording — or in a way that keeps the original pitch of the recorded sounds.

However, every new concept comes at a cost, for two reasons:

- It adds an unfamiliar concept that users must learn.
- It potentially interacts with every other concept in the system. As concepts are added to a system, the complexity of the system rises not linearly, but multiplicatively.

Therefore, additional concepts should be resisted, and admitted into the conceptual model only when they meet functional requirements for the application and their extra cost can be minimized through good user interface design. Remember: less is more!

4.3.3 HIDES CONCEPTS THAT ONCE WERE NECESSARY BUT NO LONGER ARE

Sometimes new technology automates a feature or function that formerly required explicit user action, thereby eliminating the need for people to invoke or control it. In such cases, software designers can simplify the application by eliminating the corresponding concepts from the application's conceptual model and hence from its user interface.

For example, most automobiles built before 1930 required drivers to control the mix of fuel versus air that was injected into the engine's pistons. The control was called a "choke". Drivers pulled the choke *out* or pushed it *in* to adjust the fuel/air mix and the running of the engine. Automobile companies introduced automatic chokes in the 1930s and fuel-injection systems in the 1950s. Both eliminated the need for drivers to monitor and adjust the fuel/air mix, so the choke (i.e., fuel-air mixture) control vanished as a part of the conceptual model of the common automobile.

In the realm of familiar software applications, early word-processing systems had a concept of a spell-checking operation, which users explicitly invoked when they wanted to check their spelling throughout a document. In modern document editors, spell-checking occurs automatically and constantly in the background unless disabled, eliminating spell-checking as an explicit on-demand operation.

When a new application's conceptual model omits concepts that were familiar to people, designers should ensure — through user research — that users are not disoriented by the concept's disappearance. Usually, they won't be. People like things simple; the simpler, the better. For example, few people who remember the choke control in automobiles miss it.

However, when familiar concepts are removed from a conceptual model, the documentation describing the model — for both developers and users — should explain the absence of the concept, to make clear that the concept was intentionally omitted.

4.3.4 SHOWS RELATIONSHIPS BETWEEN CONCEPTS

Enumerating the objects, attributes, and operations for a task-domain allows designers to notice relationships between objects. Making these relationships explicit in the conceptual model can make the application easier for users to understand.

A Common Relationship: Specialization

Different objects in a task-domain may have the same operations and attributes. This suggests that those objects may actually all be variations of one type of object, or sub-types of a common parent object. In such a case, objects in a conceptual model can often be organized in a *specialization* or *type/sub-type hierarchy*, in which certain conceptual objects are *specializations* of others[7]. Alternatively, similar objects can be represented as a single object-type with an attribute indicating which of the possible variations they are.

Whichever approach is used, making the similarity relationship explicit in the conceptual model can help users learn more easily about all the conceptual objects. Most people can understand the idea of specialization: for example, a **checking account** is one type of **bank account**, a **book** is one type of **product** a store might sell, and a **cookbook** is one type of **book**. Furthermore, designers can take advantage of the commonalities across similar objects: they can use the same interaction design for common operations and the same design for viewing and setting object attributes.

For example, consider a drawing application that provides functions for drawing both **rectangles** and **ellipses**. If both **rectangles** and **ellipses** have similar attributes and operations, once a user understands one, they also understand the other. As a specific example, once users learn that they can constrain **rectangles** to be squares, they should be able to immediately assume that they can similarly constrain **ellipses** to be circles. Such consistencies result in a conceptual model — and an application — that appears to have fewer distinct concepts, is simpler and more coherent, and is more easily mastered.

Another Common Relationship: Containment

In many applications, objects are also related by a second hierarchy: *containment*, in which some objects can *contain* other objects. For example:

- a **print-queue** *contains* **documents** about to be printed;
- a **shopping cart** *contains* **products** a customer wants to purchase;
- an **email folder** *contains* **email messages**;
- an **organization** *contains* **employees**.

[7] Specialization is also a common relationship between objects in object-oriented analysis, a common early step in object-oriented programming. However, type/sub-type relationships in conceptual model design only concern objects that the application exposes to users.

If objects in a conceptual model are related by containment, the model should indicate whether the containment relationship is mutually exclusive or not, that is, whether an object can be in multiple containers at once or only in one container at a time (see Chapter 7: Optional Modeling.)

Other Object Relationships

A conceptual model for an application can also specify other relationships between objects if they are important to the tasks it is intended to support. Examples of relationships sometimes made explicit in conceptual models:

- ***Whole/part:*** This is similar to containment, but not exactly the same. For example, a bus contains passengers, but they aren't part of the bus. The exhaust pipe, on the other hand, *is* part of the bus. Sometimes, it is hard to see the difference, e.g., a chapter is both contained in a book and part of it, and a paragraph is both contained in a document and part of it. Thus, part/whole is similar enough to containment that some designers model them as the same relationship (see also Chapter 6: Essential Modeling).

- ***Source/result:*** One object may be a source, e.g., a datafile used in an analysis, while another is the corresponding result, e.g., the datafile coming out of the analysis. Links and their referents can be considered related in this way, as can "raw" digital images and the cropped and adjusted photographs that are made from them.

- ***Task/sub-task:*** One task may be a part of a larger task, e.g., opening a bank account consists of several steps, each of which can be regarded as a task on its own. This relationship is between operations, rather than between objects. This relationship often "comes for free" when a designer lays out the type/sub-type and whole/part relationships between objects.

Relative Importance

Finally, concepts in a conceptual model — objects, attributes, and operations — are related to each other in *importance*. Some concepts are encountered by users more frequently than others. For example, closing a checking account is an infrequent operation compared to, say, entering a transaction into an account. The relative importance can be used to focus the design: it is more important to make frequent operations easy, even at the expense of less-frequent ones. In contrast, less frequent operations can involve more work, but should be easy to remember from one time to the next.

4.4 EXAMPLE OF OBJECTS/OPERATION ANALYSIS

As a concrete example, let's examine an objects/operations analysis for a simple office calendar application. The objects, attributes, operations, and relationships might be as follows:

- **Objects:** It would include objects such as **calendar**, **event**, **to-do item**, and **person** (see Table 4.1). It would *exclude* non-task-related objects like *buffer*, *dialog box*, *database,* and *text-string*.
- **Attributes:** It would make sense in a conceptual model for a **calendar** to have an *owner* and a *default focus* (day, week, month). An **event** could have a *name*, *description*, *date*, *time*, *duration*, and a *location*. A **to-do item** could have a *name*, *description*, *deadline*, and *priority*. A **person** could have a *name*, a *job-description*, an *office location*, and *phone number*. **Events** should not have something like a *byte-size* as a user-visible attribute, because that is implementation-focused, not task-focused.
- **Operations: Calendars** could have operations like *examine*, *print*, *create*, *change view*, *add event*, *delete event*. **Events** would have operations like *examine, print,* and *edit*. **To-do items** would have more-or-less the same operations as **events**. Implementation-related operations like *loading* databases, *editing* table rows, *flushing* buffers, and *switching* modes would not be part of the conceptual model.
- **Relationships:** **Meetings**, **vacations**, **birthdays**, and **personal appointments** could be modeled as objects that are sub-types of **event**, thereby inheriting all the attributes and operations of events but perhaps having additional attributes and operations. A simpler alternative (shown in Table 4.1) is to model different types of events as values of a *type* attribute on Events, e.g., *type* = *birthday*.

Table 4.1: Object/operation analysis for a simple office calendar application.

Objects	Attributes	Operations
Calendar	owner, current focus	examine, print, create, add event, delete event
Event	name, description, date, time, duration, location, repeat, type (e.g., meeting)	examine, print, edit (attributes)
To-Do item	name, description, deadline, priority, status	view, print, edit (attributes)
Person	send email, view details	name, job-description, office, phone

4.5 CONCEPTUAL DESIGN ISSUES

A final component of a conceptual model is a list of resolved and still-open conceptual design issues; i.e., known problems in the conceptual model. If an issue has been resolved, it should be so marked (or moved to a separate list of resolved issues), and the resolution should be recorded with the issue, so newcomers to the design project can see why the design is as it is. The following are examples of Resolved and Open issues from a hypothetical server management application:

Resolved Issues

1. What is the scope of a log? Are logs global to the entire Management Console, or are they specific to Enterprises? *Resolution: In principle, logs are scoped by Enterprise, but since version 1 has only one Enterprise, version 1 can be regarded as having only one log, which is global.*

2. How is the "source" of an Event identified? *Resolution: Component ID, location.*

Open Issues

1. The log will record huge numbers of events, so it will be hard for users to find and keep track of important problem-events. One solution is to have separate logs for normal vs. problem-events, and the ability for users to delete problem-events once handled. Another possible solution is a filtering mechanism for the log, allowing users to view only small subsets of interesting events.

2. The model has a concept of an Enterprise: an enclosed, protected environment in which to services are run. One issue is whether we also need the concept of a Hardware Domain: a division of a physical server's hardware resources, upon which Enterprises run. If users need to know which physical machines their Domains have been placed on, the concept of Hardware Domains may be needed.

4.6 MAPPING FROM TASK-DOMAIN

This component describes task-to-tool mappings: how people can use an application to help carry out tasks in the application's target task-domain. As discussed in Chapter 1 (Using tools), a person with a task creates a plan for combining the operations provided by the application – together with other activities that they do themselves (including using other tools) – to accomplish their task.

This component of a conceptual model demonstrates the quality of the model by showing how easily common tasks can be performed. A conceptual model that fits its target task-domain well will have simpler plans: concepts in the task domain will map directly to concepts in the conceptual model. Plans for less-common tasks may be more complex. If the mapping shows a need for people to form complex plans in order to carry out common tasks that should be simple and easy, redesign is indicated.

One of the best ways to describe mappings is to express them as task-level scenarios (see Chapter 3) using the concepts and terminology of the conceptual model.

For example, most photocopiers are designed so that the task-to-tool mapping (plan) for making one copy of a one-page original is very simple: put the page in the original tray, push start, take the copy out of the copy tray, and take your page out of the original tray (UI hint: remind people not to forget the last step). As discussed in Chapter 3, doing a complex copying task on a simple copier will require a more complex plan.

However, in the reprographics shop of a large business, simple copying tasks might be rare. Doing the one-copy-of-one-page task on one of the commercial photocopiers may have a more complex plan: an operator might have to adjust many settings to set up the "simple" copying job.

Design of the conceptual model must be driven by the task-domain. The mapping component of the conceptual model demonstrates that the design is successful, by showing that common tasks have simple plans, and complex plans are needed only for rare tasks.

CHAPTER 5

Example

To make the ideas presented in the previous chapters more concrete, this chapter presents an example of a conceptual design built around a conceptual model. Previous chapters have provided examples of parts of conceptual models, some of which are for hypothetical simple applications and some of which are for real, more complex ones. In contrast, the example conceptual model presented here is complete – with all its parts. In fact, it uses the same format as the conceptual design documents that we prepare for client companies. Those documents and the rest of the chapter are structured as follows:

- Overall purpose
- High-level functionality
- Major concepts and vocabulary
- Task hierarchy enumeration
- Objects-operations analysis
- Resolved conceptual design issues
- Open conceptual design issues

The conceptual design is for a high-end Web-Search service intended mainly for professional information workers. It is adapted from an actual design done several years ago, so the name of the service has been changed and many details have been changed or omitted for the sake of brevity, clarity, and intellectual property protection.

CONCEPTUAL MODEL: DOCLIVINGSTONE SEARCH SERVICE

5.1 OVERALL PURPOSE

The purpose of the **DocLivingstone** service is to allow people, mainly professional information workers (e.g., librarians, travel agents, patent attorneys) to search for information on the Web in a more systematic, sustained, comprehensive, and flexible way than most search services support. Search in DocLivingstone is systematic, sustained, and comprehensive because search projects span multiple individual searches and even multiple usage sessions, retaining the information collected until it is no longer needed. Search in DocLivingstone is flexible because DocLivingstone accepts as search input not only keywords, but also documents and web URLS.

5.2 HIGH-LEVEL FUNCTIONALITY

The overall design consists of the following important conceptual design decisions:

- DocLivingstone users can create Search projects (called **Explorations**), which persist across individual searches and use-sessions. Explorations can be started, worked on, set aside, returned to, and resumed later. A user may have several Explorations at any point in time, and may shift from working on one to another at will. Explorations hold found information that the user wishes to retain. They can be deleted when no longer needed.
- As input to a search, users can provide text (search terms or prose), webpage URLs, textual documents, or a combination of these.
- Documents returned by searches (formerly called Search Result Items) can be organized in different ways, e.g., as an ordered list (table) of individual found documents (like results in most search engines), or by common topics extracted from the documents.
- Users can refine searches by indicating that certain found documents are especially relevant. DocLivingstone uses this to revise the search, thereby modifying the search results.
- Specialized **Explorations** (e.g., business travel, patent search) are provided for certain common use-cases. These guide users through the steps of conducting a search, and provide results-displays and analyses customized for the type of **exploration**.

5.3 MAJOR CONCEPTS AND VOCABULARY

This section defines terms for the service. The following terms should be used consistently in the service and all documentation about it.

- **DocLivingstone:** The service itself. Needed because users need a way to refer to it.
- **Exploration:** A container for a user's search-work in pursuit of an information goal and the results it generates. Contents of **explorations** include Search input, topics, results lists, found documents (aka results items), notes, and analysis results. An **exploration** is a way of organizing a search, refinements, and the resulting data according to the high-level goal a user is trying to achieve (similar to *projects* in other

applications). An **exploration** can contain multiple Search Results Lists (see below). Each Search Results List documents the input on which it is based.

- **Exploration List:** A list kept by the service of a user's current **explorations**.

- **Search Input:** Information the user supplies to an **exploration** to indicate what s/he is looking for. Input to a search consists of text, which can be supplied in a variety of different ways: typing search terms, typing full sentences or paragraphs, specifying documents, indicating web-pages. Search Input may consist of one or more Search Input Items.

- **Search Input Item:** A single piece of input to a search, provided in one of the several possible ways: typed text, document, URL.

- **Search Results List:** A list of **documents** found by a search, rank-ordered by match score, from highest match to lowest.

- **Document:** Formerly called "Search Result Item". An item found by a search. **Documents** can be any of several media types: patent, patent application, advertisement, magazine, journal, newspaper, book, website, blog. The subject of a Document is separate from its media type.

- **Media Type:** The medium of a **document** by a search. Media types include: newspapers, magazines, advertisements, journals, books, websites, blogs (a special type of website), patents, and others. A **document's** media type determines the attributes it is assumed to have, e.g., blogs have contributors, dates, etc.; patents have classifications, assignees, etc.

- **Content Category:** Pre-defined categories into which all textual content is categorized. The pre-defined categories are: technology, business, entertainment, sports, science, medical.

- **Relevance Feedback:** Users can provide the service with feedback on the actual relevance of a given **document** returned by a search. This information is used by the service to revise the search criteria to give higher weightings to the attributes in found items that the user marks as highly relevant, and then initiate a new search.

- **Topic:** A phrase derived from common content in a list of Documents returned by a search, which serves as a title for that content.

5.4 OBJECT-OPERATIONS ANALYSIS

5.4.1 FORMATTING CONVENTIONS

The objects and operations in the service have a hierarchical structure. As an aid in presenting this hierarchy as well as the relationships between operations and objects, the following typographical conventions are followed.

- Objects names are **bold**.
- Each object has "slots" for subtypes, attributes, and operations. In some objects, some of these slots are empty, that is, some objects have no subtypes, or no attributes, or no operations.
- Objects that are contained in other objects indicate their container object type in a Container slot.
- Objects that have parent-types inherit attributes and operations from their parent unless they redefine the attribute or operation.
- Some operations have sub-actions, or variations. They are listed below the main operation.

5.4.2 OBJECT-OPERATIONS HIERARCHY

- Exploration
  - Subtypes
    - *none*
  - Attributes
    - *Name: text*
    - *Description: text*
    - *Create date: date & time*
    - *Modify date: date & time*
    - *Results: list of* ***Documents***
    - *Alert:*
      - *Set: ON/OFF*
      - *Checking Frequency: enum {hourly, daily, weekly, monthly}*
      - *Alert Method: enum {markUserTask, email}*
    - *Notes: text*
    - *Operations: list of past operations in this Exploration, latest first*

  - Operations
    - *Search for Documents matching specified Search Input*
    - *Review Search Results*
    - *Refine Search*
    - *Undo last operation*
    - *Redo last undone operation*
    - *Copy*
    - *Delete*
    - *Set Alert (to recheck periodically and notify user when something new is available)*
      - *Turn ON/OFF*
      - *Set Checking Frequency (freq)*
      - *Set Alert method (method)*
    - *View Alert*
    - *Save (only for* **Explorations** *that have not yet been saved)*

- **Exploration List** (one per user)
  - Attributes
    - *Username: text < valid username >*
  - Operations
    - *Clear*
    - *Expand* **Exploration**
    - *Contract* **Exploration**
- **Search Input**
  - Attributes
    - *Items: List of Search Input Items*
  - Operations
    - *Submit*
    - *View*
- **Search Input Item**
  - Subtypes
    - *typed text, file, web page*
  - Attributes

        * *Content: text*
    - Operations
        * *View*
- **Typed Text Search Input Item** (subtype of Search Input Item)
    - Attributes
        * *TBD*
    - Operations
        * *View: display of text*
- **Filed Search Input Item** (subtype of Search Input Item)
    - Attributes
        * *Filename: text*
    - Operations
        * *View: display of filename*
- **Web Page Search Input Item** (subtype of Search Input Item)
    - Attributes
        * *URL: text*
    - Operations
        * *View: display of URL*
- **Search Result List**
    - Attributes
        * *Input Data: Search Input*
        * *Description: text string*
        * *Items: list of* **Documents**
        * *Display controls:*
            - *Category shown: enum {<subtypes of Documents >}*
            - *Number of items shown: integer*
            - *Order by: enum {relevance, popularity, <other?>}*

  - Operations
    - *View/Edit Attributes*

- **Document (aka Search Result Item)**
  - Subtypes
    - *patent, advt, magazine, journal, newspaper, book, website, blog*
  - Attributes
    - *Name: text*
    - *Datafields: [pertinent data differs by subtype]*
    - *Content Category: enum {technology, business, entertainment, sports, science, medical}*
    - *URL: url*
    - *Date: date*
  - Operations
    - *Give relevance feedback*
    - *Make input item*
    - *View Attributes*
- **Patent Document** (subtype of Document)
  - *<details omitted>*
- **Advt Document** (subtype of Document)
  - *<details omitted>*
- **Magazine Document** (subtype of Document)
  - *<details omitted>*
- **Journal Document** (subtype of Document)
  - *<details omitted>*
- **Newspaper Document** (subtype of Document)
  - *<details omitted>*
- **Book Document** (subtype of Document)
  - *<details omitted>*

- **Website Document** (subtype of Document)
  - *<details omitted>*
- **Blog Document** (subtype of Document)
  - *<details omitted>*
- **Topic**
  - Attributes
    - *Label: text (not user settable)*
    - *Contents: list of* **Documents**
  - Operations
    - *View Content documents*

5.5 MAPPING: TASK HIERARCHY ENUMERATION

This section enumerates the tasks the **DocLivingstone** service is intended to support. These tasks are common to most or all of the top-level consumer searches described in the previous sub-section. Major tasks are broken down into sub-tasks, some of which are task-steps, while other sub-tasks are alternatives.

- Provide input to search
  - Type input keywords or description
  - Provide input file
  - Provide input URL
- Start search
- Review search results
  - Navigate through Themes and Documents resulting from a search
  - Open a resulting Theme folder to view the Documents in it
  - Open a specific found Document to view its content
- Act on resulting Documents
  - Save Document in Exploration’s Saved Documents list
  - Mark Document as relevant to refine search results
  - Mark Document as not relevant to refine search results

- View Exploration's Saved Documents list
  - Delete Document from Saved list
- Alter Exploration Details
  - View Exploration Details
  - Change Exploration Type
  - Edit Exploration Input
- Start a new Exploration
- Switch to different Exploration
- Maintain Explorations
  - save Exploration
  - name/rename Exploration
  - provide description
  - delete Exploration(s)
- Set Alert on Exploration
- View Alert Notification

5.6 RESOLVED CONCEPTUAL DESIGN ISSUES

1. Users can provide web URLs as input to a search. Does a URL specify a web *page* — i.e., only the specific page having that URL — or an entire web *site*? If a URL means a web *site*, then how does the service know where the site ends? How many links away from the given URL does it scan? *Resolution: At least in the first release and perhaps forever, a URL provided as input to a search specifies only a single page. The search input mechanism does not follow links outward from the specified page.*

2. Are patents and patent applications the same concept or different concepts? *Resolution: Patents and patent applications will be treated as the same concept. The only difference between them is that patent applications have null values for certain attributes, e.g., Grant Date.*

3. Within an Exploration, users can refine a search in several different ways. Each refinement produces a new set of results. Does the service keep the results of every refinement, or does it keep only the results of the latest refinement? Can users get back to the result of any Search they have done in a given Exploration? *Resolution:*

Only the results of the last search are kept in a given Exploration. Users refine the Search by indicating relevance of items or by turning result-documents into Input Items. However, actions taken in a Exploration are saved, so in principle, users can back out of them one at a time, using something akin to UnDo.

5.7 OPEN CONCEPTUAL DESIGN ISSUES

1. If a user chooses not to organize different search projects into separate Explorations (i.e., ignores the concept of Explorations, treating DocLivingstone as a simple search tool), everything the user does after initiating a Search goes into a default Exploration. This includes providing relevance feedback on items, performing analyses, and other actions. However, if the user provides new Search input, the current design starts a new Exploration. It is unclear whether — or under what conditions — the first Exploration will be saved or overwritten by the new one. One possibility is that Explorations are not saved unless a user either a) explicitly saves them, or b) implicitly indicates that the Exploration is save-worthy by, e.g., renaming it something other than the default name. Also, a default Exploration could be saved across sessions, or it could be forgotten. If it is no trouble to save it automatically, it would be better to do that than to require users to explicitly save their work.

2. In Search Results, if a user marks a resulting document as a good result, the service uses that to adjust the weights of its selection and ranking criteria, so the Search Results list may change. However, that does not alter the search input for the Exploration. Thus, a given Exploration has *state* that is not reflected in its Search Input. Is that state visible anywhere to users? If not, it may seem mysterious that two people can provide the same Search Input, yet have different Search Results.

CHAPTER 6

Essential Modeling

Modeling a task-domain comprehensively and coherently is difficult. Modeling it comprehensively, coherently, and *simply* is even harder. Modeling it comprehensively, coherently, simply, and in a way that makes sense to people who want to do things in the task domain is *very* difficult. This chapter provides advice on how to model task-domains in ways that meet all of these goals.

It begins with some Basic advice on performing object/operations analysis. Following the basics, this chapter discusses common conceptual design issues that often cannot be resolved early in design, but *must* be resolved before a conceptual model — and the application embodying it — will make sense.

The next chapter (Chapter 7) covers additional issues that commonly arise during conceptual design, but that are *optional*: designers can ignore them and still have a coherent conceptual model and application.

6.1 BASIC OBJECT/OPERATIONS ANALYSIS

Designers new to conceptual models often have trouble with objects/operations analysis: they encounter hurdles in deciding: a) whether concepts should be modeled as objects, operations, or attributes, and b) which object each operation acts upon. Here are some tips to help designers over some of these hurdles and perform object/operations analysis successfully.

6.1.1 ASSIGN OPERATIONS AND ATTRIBUTES TO OBJECTS

Any objects in the model are there because users can *do* something with them. Conversely, anything users can *do* must be done to *something* — some *object*. There should be no operations that are not assigned to one or more objects.

For example, in a conceptual model for a bank ATM, a *view balance* operation would be assigned to the user's **bank account** object.

6.1.2 ASSIGN OPERATIONS TO THE APPROPRIATE OBJECT(S)

Sometimes it is unclear which object an operation applies to. For example, consider the objects and operations for a library application. One operation is searching for a specific book (by title, author, etc.). It is tempting to say that this Search operation is an operation on a *book*. But think about it for a second. It's true that the operation searches *for* a book, but what is the Search operation searching? It does *not* search a *book* — that would be a *different* operation, one to look *inside* a specific book for a passage. Instead, it searches the library's *catalogue* for a book. That means that the library's *catalogue* is an object in the application, with Search (for book) as one of its operations.

However, sometimes it truly is unclear to which object an operation belongs. For example, imagine a conceptual model for an online calendar application. Some calendar applications allow users to create and manage multiple calendars for scheduling different types of events, e.g., personal *versus* work-related. In such an application, what is the object for the operation that creates a new calendar? Is *Create* an operation on the **Calendar** object, or is *Create Calendar* an operation on the **calendar application**? The latter approach is more consistent with the solution to the library catalogue problem discussed above. However, many designers would opt for the former approach — making *Create* an operation on Calendar — because software designers and developers often assign operations that create a new object to the object to be created (e.g., Calendar.new). Designers can choose to do it however they want, but they should be consistent throughout the conceptual model and the user interface designed from it.

6.1.3 DECIDE HOW TO MODEL SIMILAR OBJECTS

If several objects have similar operations and attributes, the conceptual model should reflect that similarity and not treat the objects as if they were not related. For example, a conceptual model for an office calendar application might have several different types of events: meeting, birthday, vacation, personal appointment, etc. These different types of events would share many operations (e.g., create, view, edit attributes, delete), and attributes (e.g., name, description, time, date). One approach is to model them as sub-types of some generic object, e.g., include an *event* object with all the common operations and attributes, and sub-type objects (e.g., meeting, birthday), each with any relevant extra operations and attributes. An alternative approach is to include only the generic object type (e.g., event), and model the variations with a *Type* attribute that indicates the type of meeting it is (e.g., Type: birthday).

How do designers decide between these two ways to model object similarity?

If the similar objects are *so* similar that their operations and attributes are identical or nearly so, it makes more sense to model the sub-types using a *Type* attribute. However, if the similar objects have overlapping but different operations and attributes, it makes more sense to model them as objects that are sub-types of the generic object-type.

For example, compare Table 4.1 (Chapter 4) with Table 6.1 (above). Table 4.1 represents a conceptual model for a simple office calendar, with different types of **events** modeled as values of a *type* attribute on **events**. In contrast, Table 6.1 represents a conceptual model for a slightly more complex office calendar, in which different types of events are modeled as objects that are sub-types of **event**. This conceptual design would be preferred if the different Event-types have significantly different attributes and operations.

A second reason for using object sub-types rather than a *type* attribute is if designers need to represent sub-categories of any of the sub-types. For example, if a calendar application has **meeting** as one type of **event**, and the calendar needs also to provide different types of **meetings** — e.g.,

Table 6.1: O/O analysis for an office calendar with different Event-types modeled as subtypes.

Objects	Attributes	Operations
Calendar	owner, current focus	examine, print, create, add event, delete event
Event (types indented below)	name, description, date, time, duration, location, repeat	examine, print, edit (attributes)
Meeting	host, invitees, dial-in number, access code	announce, cancel
Vacation	start date, end date	
Personal appointment	private	
Birthday	surprise	
To-Do item	name, description, deadline, priority, status	view, print, edit (attributes)
Person	send email, view details	name, job-description, office, phone

all hands, **manager-employee one-on-one**, **design team**, etc. — then it is necessary to model the types of **event** as object sub-types rather than as attribute values, so **meeting** can also be broken down into *its* sub-types (either by sub-types or by attribute).

6.1.4 DECIDE WHETHER TO INCLUDE THE GENERIC OBJECT TYPE

Even if designers decide to model similar objects as sub-types of a generic object-type, they are under no obligation to include the generic object-type in the conceptual model. For example, a conceptual model for a home banking application could include **savings accounts**, **checking accounts**, and **time-deposit accounts** without including (i.e., exposing to users) the concept of a generic **bank account**.

Similarly, a conceptual model for an office calendar could include **meetings**, **birthdays**, and **vacations** without including the concept of an **event**. Of course, the application's developers will probably include the generic object-type in their *implementation* model to simplify the coding, but that doesn't mean the generic object has to be exposed to users.

6.1.5 DECIDE WHAT TYPE OF VALUES AN ATTRIBUTE HAS

An attribute must have a value. The value has a type, e.g., text, number, date, day-of-week, month, etc. For example, Name attributes and Description attributes would have text as their value. A Date attribute would have a date of the year as its value. An bank account's Owner attribute would have a Person (i.e., an object) as its value. A meeting Invitees attribute would have a *list of person* objects as its value.

Thus, objects in a conceptual model can be the values of attributes in the model. If designers add an attribute and notice that its value is an object that is not in the model, they must either add that object to the model's object list or change the attribute not to have that object as its value. For example, if the designers of a calendar application want events to have an Invited attribute that lists Person objects, then the conceptual model must include a Person object. If the designers don't want to include a Person object in the model, they have to change the Invited attribute to have as its value a list of email addresses for people not necessarily known to the application.

6.1.6 DECIDE HOW DETAILED TO BE IN MODELING COMMON OPERATIONS

Some operations are covers for collections of related sub-operations. Designers have a choice when creating a conceptual model whether to include only the cover operation, both the cover and the detailed sub-operations, or only the detailed sub-operations.

For example, in a conceptual model for a document editor, documents (an object) might have an Edit operation. But users can edit many things in a document: paragraphs, words, fonts, paragraph styles, margins, etc.

Similarly, in a calendar conceptual model, events (an object) presumably have attributes (e.g., Name, Description, Start Time, End Time, Date, Repeat) that could be modeled by simply giving events an Edit Attributes operation, or it could be modeled by providing Edit Name, Edit Description, Edit Start Time, etc.

How designers should handle such a situation depends on how detailed the conceptual model needs to be to be for those working *from* it — e.g., programmers, technical writers, marketing copywriters — to be able to understand and use it effectively.

6.1.7 INCLUDE ALL TASK-RELEVANT OPERATIONS

Any task-domain operation that the application will support should be included in the objects/operations analysis. This includes operations that may not require any user-action other than looking at the user interface.

For example, telephones in hotel rooms and mobile phones often have an indicator light that is ON or an icon that is displayed if any messages have been left. Determining if any messages have been left requires no clicks or commands — only glancing at the phone. Nonetheless, it is an operation that should be included in the device's conceptual model. One could imagine *other* message devices that required pressing buttons or dialing a message-center to check for messages.

Similarly, most personal computer operating systems provide a way for users to find out what time of day it is. In Unix and Linux systems operated via command-line shells, users issue an explicit command to do this. In most window-based OSs, users need only move their eyes to where the time is continuously displayed on the screen. The UI for checking the time happens to be a "noclick" UI [Isaacs and Walendowski, 2001]. Conceptual models are independent of any specific user interface, so even though it may have a zero-click UI, checking the time of day is a user operation, so a conceptual model for an OS should include it as an operation.

6.1.8 PART-OF AND CONTAINMENT RELATIONSHIPS NEED NOT ALWAYS BE DISTINGUISHED

Even though part-of and containment are usually conceptually distinct, sometimes it just isn't worth the trouble to distinguish them. If users need not understand the difference, designers can safely treat part-of and containment as the same relationship.

Furthermore, in some situations, they actually cannot be distinguished. Are grains of sand in an hourglass contained by the hourglass or are they part of it? They are both. Are figures in a book chapter *contained* by the chapter or are they part of it? Again, both.

6.2 SUPPORTING LEARNING

Regardless of the subject matter of an application, its users will need to learn its conceptual model, at least partially. The more completely they learn it, and the more accurate their mental representation of the model is, the better. How can designers construct a conceptual model so as to facilitate learning, i.e., make it faster and more accurate?

6.2.1 METAPHORS

As discussed in Chapters 3 and 4, a conceptual model can be designed to reflect or reference another (presumably well-known) conceptual model. The referenced conceptual model is often called a "metaphor." For example, the personal computer *desktop* with its *files* and *folders* is a conceptual model built using as a metaphor the conceptual model of a physical paper-based office with its *desktops* and *files* and *folders*. (The electronic objects are said to be represented by *icons*, which is *itself* a metaphor based on religious imagery.) E-mail is based on physical memos, with their subjects, dates, to- and cc-addresses. Spreadsheets have arrays of cells, analogous to the cells of paper spreadsheet. Letting one conceptual model stand on the shoulders of another is a powerful way to accelerate learning and provide support for use.

However, analogies and metaphors are limited and inexact [Halasz and Moran, 1982, Johnson, J., 1985, 1987]. Users may be misled into expectations that the application does not fulfill. For example, the amount of material in a real-world physical folder is usefully indicated by the folder's size and weight. In contrast, the amount of material in an electronic folder is *not* indicated by the size of the folder's icon on the electronic desktop, and the icon has nothing analogous to weight, e.g., dragging the folder doesn't become more sluggish the more it contains.

Designers must take care in conceptual design to ensure that the objects, attributes, and operations of the reference conceptual model (the metaphor) are carried over into the application's conceptual model. Where they aren't, users must be made aware of the discrepancies.

6.2.2 PROGRESSIVE DISCLOSURE

When an application is rich but many users will have make use of only a small subset of the supported task-domain, designers often provide more than one user interface to the application. For example, on some photocopiers, a few simple tasks are supported by a small subset of the conceptual model: make one copy, make a small number of copies, make the copies two-sided. However, with a photocopier that can shift images on pages, provide covers, staple, and bind, these extra capabilities are often hidden until revealed by opening a covering panel. When the copier user interaction is on a display, "Advanced features" buttons display more detailed controls.

This is called "progressive disclosure," reflecting the notion that as the user's tasks progressively become more complex, the application discloses more of its conceptual model [Johnson et al., 1989, Johnson, J., 2007]. However, a better way to think of this is that now there are *two* conceptual models, with mechanisms for moving between them. In such cases, it is very important that the simpler model be — and be exposed as — a clear and exact subset of the advanced model. So, objects and operations from the advanced conceptual model should be sharply delimited so that they do not leak into the simpler conceptual model. It is particular hard on users when the advanced conceptual model requires that simple tasks be thought of differently than they are in the simple conceptual model. When this happens, users can easily become confused about *both* conceptual models.

6.2.3 COMPONENT MODELS

Large multi-function applications often have complex conceptual models. One way of addressing this complexity is to separate it into components, each of which is modeled more-or-less separately.

For example, the conceptual model of a car might include component models for operating the car (steering wheel, pedals, etc.), for adjusting the car's amenities (windows, seats, heater, radio, etc.), and for the engine. In the user interface, some or all of the components — and therefore some of the component conceptual models — can be hidden until needed behind progressive disclosure mechanisms (e.g., the car's hood, which conceals the engine), or they can all be available constantly.

6.2.4 SURROUNDING MODELS

All applications function and are used in some computational context or environment. The environment provides a surrounding conceptual model that users may be aware of, which may impact the conceptual models of applications that run *in* that environment. Here are some examples:

Time

Most computer systems represent time in a very technical way (e.g., the number of seconds since midnight, Jan 1, 1900). However, an application would represent times-of-day in a more user-friendly way. For instance, a calendar might indicate that a meeting starts at 10:00 am Pacific Daylight Time on a particular day. The conceptual model would include something like:

- a **moment-in-time** is independent of location;

- **time-zones** connect **moments-in-time** to places;
- **time-intervals** (**year**, **month**, **week**, **day**, **day-of-the-week**, **hour**, **minute**, **second**) depend on **time-zones**.

The application would have to integrate these time-related concepts with its other concepts. For example, in a calendar, although a meeting starts at a moment in time, the time-labels for that start-time will differ for participants in different **time-zones**; it may even be on different days, or even years. Are users of the application only interested in what the time is in their time-zone, or do they want to know how other participants see it? What time will that be in New York City? Is that a holiday in Bangalore?

Files

Many applications borrow concepts from the operating systems on which they run. One of the most common is ***files***.

For example, in one company's application for creating **maps**, a key concept in the conceptual model was **table**, the aggregated data defining a map. Historically, the application had used spreadsheet files to represent **tables**, so users could edit them with spreadsheet applications. Thus, the map data files were modeled using both operating system concepts (**files**) and application concepts (**tables** representing **maps**). The designers felt that because these data **files** were accessible via the operating system, the application did not have to provide its own access to them. Users had to open and edit spreadsheet **files**, with no support in the application for manipulating **tables**. Thus, the application borrowed concepts from its environment.

Security

In applications that operate in the Internet (and few do not these days), access to objects must be carefully managed. Sometimes applications run entirely within protected environments (e.g., a secured machine), so security is not an issue for their conceptual model. More often however, Internet applications allow users to access objects remotely. For example, events on a shared calendar may be accessible by some and not others.

The computation platforms on which applications run provide powerful mechanisms for limiting access, e.g., access control lists on files, encryption on passwords. However, the concepts comprising these access-mechanisms are rarely the right ones for describing how security works in the application's task-domain. For example, access in a calendaring domain should reflect something like:

- access is not given to **login names** or **machines**, but rather to **projects** and **members**.
- access is not on **files**, but on ***events*** that occur in ***calendar-overlays***.
- the possible access attribute-values are not *read* vs. *write* (as for **files**), but rather *show-existence-only*, *show-title-only*, *show-details* (controlling how much others can see).

The point of these three examples (time, files, security) is that the conceptual model of an application can be made easier or harder to learn by borrowing concepts from the conceptual model(s) of the platform on which it runs, depending on whether the users understand the platform's model.

External physical objects

Another important set of concepts that is sometimes overlooked in building conceptual models are real-world objects — objects that are external to the application. If the application interacts with its users in manipulating such objects, then describing the application requires modeling them.

For example, a photocopier carries out jobs in which it scans "original" documents, feeds blank "supply" paper, and produces documents that are copies of the original. A conceptual model for a photocopier must include documents and paper. It must also include "areas" of the paper path for clearing paper jams. It also requires toner (ink) cartridges to be added periodically, so the conceptual model must include those. Finally, photocopiers also require oiling and other maintenance functions, but only maintenance personnel would need conceptual models containing maintenance tasks and concepts.

6.2.5 OBJECT-ORIENTED VS. TASK-ORIENTED USER INTERFACES

An application exposes its conceptual model to users through its user interface. An application's user interface can expose its conceptual model using either of two very different approaches to interaction design: *object*-oriented or *task*-oriented.

Object-oriented interaction

In many user interfaces for desktop software applications, the objects, attributes and operations of the application's conceptual model are exposed directly. For example, the titles on the menu bar (or first layer in the phone tree) could be the major concepts of the application; the menu items (or the second layer in the phone tree) could be the operations (see Fig. 6.1). Attributes could appear in either place, modifying the objects or the operations. Users have a sense of direct access to the objects and operations reflecting the conceptual model and can read and modify them "directly." Task-to-tool mapping is therefore directly supported.

Learning the conceptual model is constantly reinforced while using the program: it is immediately available by looking through the labels on menus and menu items, and by reading pop-up tooltip that are associated with the menus. Documentation is often organized around objects and operations, thus further reinforcing the conceptual model.

Task-oriented interaction

An alternative way of organizing conceptual model and applications focuses on the *tasks* (i.e., operations) that the application supports, rather than the conceptual *objects*. Some such application designs define the steps for achieving each task and provide mechanisms ("wizards") that guide users through tasks to completion. Task-oriented applications are useful for walk-up-and-use situations,

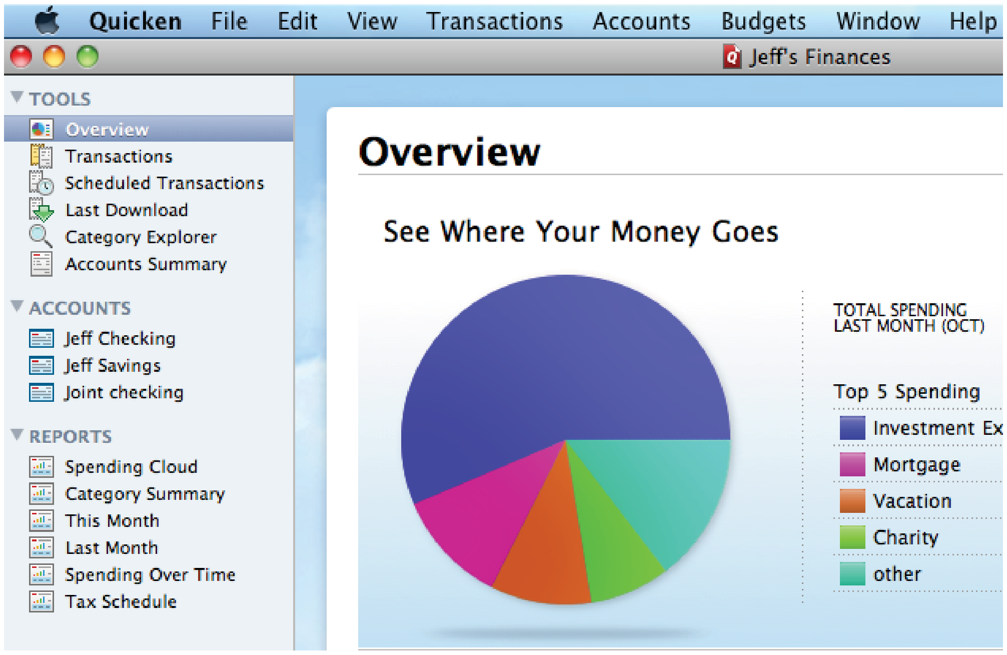


Figure 6.1: Typical desktop application has object-centric user interface.

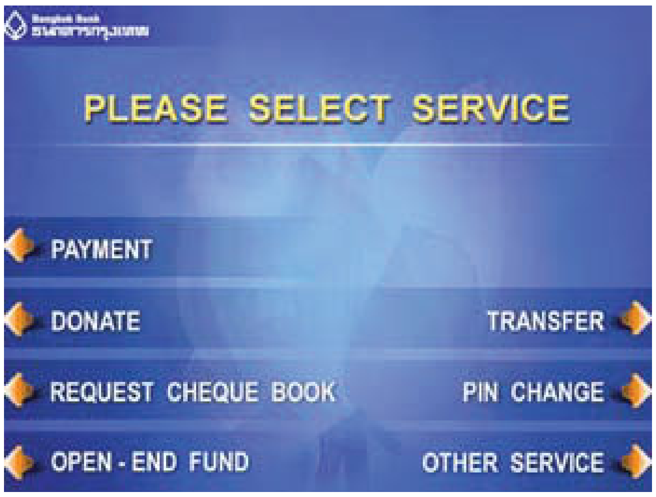


Figure 6.2: Typical ATM has task-oriented user interface.

in which users cannot be assumed to have much or any training in using the application: just choose a task and follow the instructions.

For example, most bank automatic teller machines (ATMs) let an account-holder select and follow wizards. The tasks may include: review balance, withdraw cash, deposit funds, and transfer funds between accounts (see Fig. 6.2).

6.3 CONCEPTUAL MODEL VS. USER INTERFACE

Sometimes it is difficult to determine whether an issue is the concern of the conceptual model or of the user interface. Two common but difficult cases are described below.

6.3.1 VIEW VS. FOCUS

The conceptual model is different from the user interface; the user interface exposes the conceptual model. So anything concerning interaction *mechanics* is not in the conceptual model and should be left to the user interface.

However, the line between conceptual model and user interface can be fuzzy. For example, most calendar applications allow users to choose whether the calendar presents events for one day, one week, or one month; this is often called choosing a calendar "view." It may seem that the view concerns presentation and so should be part of the user interface, not part of the conceptual model.

In fact, however, the word "view" is just a poor name for the concept. An electronic calendar has far too many **events** for users to see at once, so it must provide a way for users to focus the *scope* of the dates the calendar presents. The **focus** could be based on a variety of criteria: the desired time period (day, week, month, year), which time period (this week, next week, October, November, 2013), the type of event (meetings, birthdays), the people involved (me, Aunt Agnes), etc. The **focus** can affect not only which **events** are seen, but also which are acted upon (e.g., deleted). Thus, instead of having a view, the calendar can be (re)conceptualized as having a **focus** that controls what the calendar shows or manipulates.

In cases like this, designers should look for a conceptual object that is being hidden or mis-cast by sloppy labeling. One way to test this is to determine whether the original or altered conceptual model can be presented by a user interface that did not permit "viewing". For example, the concept of **view** would not make sense if presented through a voice-based user interface, but the concept of **focus** would.

6.3.2 INTERACTIVE CONCEPTS

Another challenging case of separating the conceptual model from the user interface occurs when an application is being designed to support tasks from a task domain that is fundamentally interactive: visual, auditory, or tactile.

For example, the task domains of graphical information system (GIS) applications include manipulation of maps, and maps include concepts that are typically considered part of the user interface: regions, locations, colors, symbols. Similarly, Web design tools and document editors manipulate layout concepts (e.g., inline, floating), distances (absolute: 200 pixels; relative: 25%).

The concepts in a conceptual model should be those that can be manipulated through *any* sort of user interface. For example, maps might have insets, and insets can be conceived of as having a *scale* attribute to provide for different amounts of detail in the same map. Of course, a user interface for maps having such a conceptual model would have to provide ways to *set-the-scale* on **insets**, which could be achieved with a two-finger swipe on a touchpad.

6.4 OBJECT IDENTITY

In applications that share objects between locations or users, the conceptual model should make it clear whether the sharing is achieved via copying or linking the objects. This issue arises in several different situations.

6.4.1 CONTAINMENT

A common but often blurred distinction in conceptual models occurs when objects are contained in containers: can a single object be in multiple containers at once?

For example, in Apple's e-mail client application, Mail, *messages* can only be in one **folder** at a time. Operations on **folders** include *move message to folder* and *delete message*. In contrast, Apple's iPhoto application allows **photos** to be in many **albums** at once, and operations on **albums** include *add photo* and *remove photo*. Importantly, removing a photo from an album does not remove it from other albums in which it has been placed. iPhoto also has a **library**, which contains all the **photos** and has its own *deleting photo* operation, which *removes* the **photo** from the **library** *and* all **albums** that contained the photo.

A conceptual model in which objects can be in multiple containers at once is needed only when objects can change. For example, if *photos* cannot change, then sharing can be implemented by copying them. The copies will all be identical and will remain so, and therefore it doesn't matter whether they are identical or copies. However, in the iPhoto conceptual model, *photos* can change, e.g., they can be annotated and adjusted. Conceptually, such changes are on the *photo* everywhere it appears, not on the instance of a photo in an *album*.

When change and containment interact, designers should take care to ensure that the conceptual model is clear.

6.4.2 SYNCHRONIZING OBJECTS

A recent variant on containment is seen in systems that move and share objects across multiple computing devices. For example, people now expect calendar applications to create events that can be shared between different calendars. Similarly, the Dropbox and SugarSync services support sharing file system hierarchies.

However, the conceptual models for these applications reflect the fact that copies are being made and synchronized with each other, and that it takes time to move events and files between computers. These new models deal in recognizing intended *identity*, *change*, *content*, and *time*. A central idea in them is that human concurrent activity on computers that are intermittently connected may result in conflicts that require human intervention to resolve. For example, applications that manage calendars, contacts, and even files (e.g., DropBox) can merge conflicting changes made simultaneously by users working at different computers.

6.4.3 INHERITING ATTRIBUTES

In complex applications, objects may occur in hierarchies, e.g., type or containment. An object's attributes can be inherited from parent objects. An issue arises when a users edit an inherited attribute.

For example, imagine an application for architects to use in designing university campuses. A campus has buildings and buildings have rooms. Buildings have attributes such as *name*, *building number*, *location*, etc. A building may also specify a *color*, which applies to all its rooms. Rooms have attributes such as *room-number*, *capacity*, *phone number*, and *color*. If a room's color is changed, does that affect the color of all rooms?

This is a conceptual design issue, and so must be made clear in the application's conceptual model; otherwise many users will misunderstand the model and change data that they did not intend to change or will be frustrated by not be able to change data they want to change.

6.4.4 WHAT WORK IS SAVED, WHEN? AND CAN I REVERSE IT?

When someone edits a document in Microsoft Word, they are editing a copy. They complete their work by invoking "Save" or (rarely) "Cancel". In contrast, in MacOS when a person modifies settings in preference panes or moves files around in folders, they are working on the thing itself, not on a copy. They complete their work by simply stopping (e.g., closing the preference window). In such cases, letting users back out of actions requires recording the actions, developing the technical capability to reverse them, and providing users with a *revert* or *undo* operation.

This distinction between these two ways of working matters most when applications fail (e.g., freeze or terminate unexpectedly): is the work a person has done lost because the copy was not saved, or is it preserved because the data was changed as the user worked?

Although this distinction is relevant in almost every conceptual model, there is no accepted language for discussing it, and no common notation is used across applications and platforms to expose it. It is useful as part of conceptual design to make clear which concepts are handled in which ways, and to evaluate whether the conceptual model is coherent. If not, users will certainly make mistakes in how they think about and treat the modification of objects.

CHAPTER 7
Optional Modeling

Chapter 6 (Essential Modeling) discussed some essential recurrent topics of modeling, topics that developers always encounter when building conceptual models. This chapter discusses two additional recurrent themes:

- going meta: activity as objects;
- evolving the application.

These themes are optional: they arise in most applications, but complete designs can usually be devised without addressing them. Nonetheless, they are important: when they *are* addressed, the resulting designs may be more usable, and may be applicable to larger task-domains.

Because these topics are optional, designers may be tempted to postpone considering them until later in development, with the thought that they can be added later if desired. However, waiting can make getting to a larger, better conceptual model much harder later. An attractive alternative is to consider these topics early so as to ensure that a coherent future design can be made. A redeeming grace is that concepts needed to support these topics need not be exposed to users in earlier conceptual models. All that is required is a migration path to these anticipated futures if the application succeeds.

7.1 GOING META: ACTIVITY AS OBJECTS

Activity, the use of applications, takes place over time. History mechanisms consider activity over time as an application concept, and allow users to navigate through their activities past and present. History mechanisms are important resources for users, in handling errors, changing direction, and preserving and repeating work. This raises several issues when designing conceptual models.

7.1.1 MANAGING TROUBLE

Errors are common when people operate tools. Sometimes they are associated with a user interface: physical (mistyping, mis-pointing) or cognitive (mis-reading, misunderstanding the menus). Sometimes they are associated with misunderstanding the tool. When errors happen, users often have to go back and repair damage.

Similarly, when users choose to change directions as their work emerges, they may have to go back and make adjustments before they proceed. For example, a graphics artist might decide to change the colors in a document because their initial color choices are hard to read, or do not fit with an emerging color strategy.

When such trouble occurs, recovering from it requires doing four things, which should be included in the application's conceptual model (and delivered through its user interface):

- *Detect*: Notice that something unexpected happened. Trouble is a discrepancy between what one expected and what one detects. Expectation is expressed in terms of a conceptual model; what one encounters is also registered and understood against that conceptual model. A compounding difficulty is that a person may not notice trouble until they have executed several additional actions.

- *Diagnose*: Figure out what happened. The clearer the conceptual model, the easier this conceptual work is.

- *Repair*: Determine how to repair the problem. Can a user just change things to the intended state, or must they "back up" and then proceed, or do they have to abandon all recent work and start over from the beginning? The provision (or lack thereof) of "undo" capability is important to note for operations in conceptual models [Abowd and Dix, 1992].

- *Resume*: Proceed, but with an awareness that trouble has occurred, forcing intervening activity. For example, after undoing an operation, the history list may well show the last command was *undo*. If they then invoke *undo* again, what will be undone, the *undo* or the previous operation?

For example, when a person is laying out an electronic document, moving a shape around in the document may produce unexpected results (e.g., nothing appeared). With work, the person determines that the object is underneath another object. They undo the move, bring the object forward, and move it again. They proceed, knowing that the application has undo, which can undo an undo; step backward, which always goes back in time; and step forward. And what has the impact of the all this activity been on resource usage? If operations cost money (as they do in a networked world), did the user pay twice? Must they "clean up" (e.g., empty the trash)? The application's conceptual model should make such things clear.

These days, people expect applications to be designed with trouble in mind, i.e., to keep track of actions, indicate status, and allow errors to be repaired. Conceptual design must model and address this. Often these parts of the conceptual model are added very late, even after the application's initial release.

Sometimes adding concepts to handle trouble requires fundamental changes in how users will have to think about their work. For example:

- *Events*: To manage trouble, people must change their focus of attention to the events that have occurred in the application. This requires that the application has maintained a record of the requests the user made, the resulting operations carried out, the objects involved, the sequence in which this all occurred.

The conceptual model must be enlarged to include concepts like **invocation** (attributes: *operation*, *arguments*, *results*) and **history-event** (attributes: *invocation*, *prior-event*, *next-event*, *time*).

The details of these concepts can be complex, but they are similar across applications. One important design consideration is that the conceptual model for history should be broken into two parts, one reflecting the conceptual model of the application without history, and one reflecting the user interface through which the user is accessing it.

- *Undo:* A well thought-out, multi-event *undo* often requires answering difficult questions about the grain-size of the events that are undone. We expect a text editor to respond to each key typed; each key-stroke is its own *insert-character* operation. Yet a corresponding *undo* operation that undoes only one *insert-character* has been shown to be much worse than developing some concept of a *typing event* that reflects chunks of typing as people think of (actually, intuit) it, and providing an *undo-typing event* operation [Washizaki and Fukazawa, 2002].

7.1.2 ANTICIPATING TROUBLE

Users also knowingly choose to save the state of an application's data as insurance against trouble, support for exploring alternatives and changes in directions. For example, databases provide for creating checkpoints (for backing up) and transactions (for making a collection of changes all together or not at all). However, these capabilities are often not reflected in the conceptual model of applications that use those databases.

7.1.3 MACROS: CAPTURING WORK ACTIVITY

Work is often repetitive. For example:

- graphic designers test filtering operations by first applying them to one image, then selecting a subset of the filters to apply to many images;
- statistical analysis often requires multiple tests, winnowing of datasets, shaping of presentations, repeated over multiple datasets;
- text editors are used by writers for managing hierarchies of chapters and sections, each with many versions.

A common mechanism for supporting repeated work is "macros". Macros are user-accessible languages that enable description of sequences of activity. For example, unix provides shell scripts, MacOS provides AppleScript and Automator, and Excel provides Visual Basic.

A common mistake is to have the macro-language access applications using the details of an application's user interface. This limits the macro-language to work only with that user interface, requiring different macros when the user interface is changed.

A better approach is for the objects and operations of the macro-language to be exactly the objects and operations of the application's conceptual model. For example, **cells** are a key object in Excel's conceptual model; and **cells** are accessible in Visual Basic within Excel. Ideally, the concepts in Automator are exactly the concepts in the conceptual models of the MacOS applications.

7.2 EVOLVING THE APPLICATION

Over time, work changes, e.g., because the world changes, better methods are adopted or better tools are employed. As part of conceptual design, it is therefore worth considering how to support the evolution of an application to support changes in how the work is done. This includes exploring how work in that task-domain is *likely* to evolve, and correspondingly, how the application must evolve to support that work. It also includes recognizing that the future may not turn out as anticipated, and therefore exploring how the application can support unanticipated change.

7.2.1 MANAGED GROWTH

Most applications are created iteratively, particularly in this age of "agile" development methods. As an application grows in functionality, its conceptual model grows with it. In fact, the best way to grow applications is by evolving the conceptual model and then reflecting it in everything else: user interface, documentation, implementation.

When developers predict the structure of future conceptual models, they should take care to design migration paths that will help users make the necessary conceptual shifts. For example, if a calendar application will offer **overlays** in a future release, the current version might designed as if it had a single (perhaps implicit) **overlay**. The impact of that could be that the concept of overlays would be familiar when it is introduced.

7.2.2 ANTICIPATED GROWTH

However, even for designers, crystal balls are unreliable. It is impossible to know what the future will bring. However, it does not take a crystal ball to anticipate that change *will* come, and that designers probably can't predict exactly what it will be.

An excellent strategy is to turn to those who will know the future when they see it — the users. Provide ways for the people using the application to extend it so as to support the work they do. There are several ways to do this:

- *Construction sets*: Some applications provide objects — often called "pieces" — that are intended to be combined within the application to create composites pieces. Such applications are often called "construction sets", reflecting construction provided in some mechanical toys (e.g., Lego, Erector Sets, Tinker Toys, Lincoln Logs). For example, applications for creating illustrations provide a set of **primitive shapes** and a *group* operation for aggregating shapes to create **composite shapes**. Often composite pieces may themselves be further combined as members of even larger composite pieces; this is often conceptualized as **pieces** with the attribute *type* (value: base/composite), with the operations for making combinations working on **pieces** of all *types*. This is called "recursive" or "injective" composition. For example, most applications supporting illustration (e.g., Powerpoint, Illustrator, Omnigraffle, Pages) have injective composition.

- *Platforms*: People commonly use concepts in one application to *represent* concepts in another. For example, the conceptual model of a spreadsheet application describes *cells* and *formulas*; the task-domain is creating spreadsheets. However, a person can use a spreadsheet to represent their monthly budgeting work. The concepts of budgeting would be represented in the spreadsheet, say with columns for *budgeting categories* and *amounts.* Further, they might copy last month's budget (spreadsheet) to start this month's. Over time, they might come to know what parts of the spreadsheet change, and what parts remain constant, month to month.

  For them, the resulting spreadsheet is both a spreadsheet and a *budget*. When one task-domain is used to represent another, we say that the application used to do the representing is being used as a "platform." Spreadsheets are intended to be used as platforms. However, other programs that are not so intended are *also* used as platforms: a text-editing application is used as a platform for invoicing; a calendar application is used as a platform for scheduling; an e-mail application is used for fund-raising.

  When designing an application that is intended to be used as a platform, the task-domain is considered at two levels: the task-domain of doing representations (using the application to represent other task-domains), and the space of task-domains that can be represented. The concepts of platform are a surrounding model (see *Surrounding models* in Chapter 6) for the concepts created using the platform. The conceptual model of the application should include concepts that make the work of representing easier. For example, spreadsheet programs have rows and columns; they also have a *make-chart* operation that takes a data range, treats it as representational (data must be in a particular form), and creates a chart.

- *Extensibility*: Applications can provide for concepts to be created, not at design time by designers, but *during use* by users. For example, many text-processing applications allow users to put headings of various types into documents. These built-in *paragraph styles* reflect standard practices of typographers. Because these practices are not universal, many text-processing applications also let users create their *own* styles to reflect their typographical practices. Thus, the application is designed to be extended: the concept of *style* is extensible.

  The conceptual mechanisms for describing extensibility include making choices (e.g., selecting a value for an attribute), composition (e.g., aggregating choices in paragraph styles), and specialization and customization (restricting an attribute's range of values). The technical mechanisms for implementing extensibility, however, can include things that are not within the conceptual model of the application at all. In particular, many applications permit extension through programming (e.g., scripting) in either application-specific formalisms or general-purpose programming languages. For example, in describing webpages, Javascript can be used to extend the behavior of the foundational HTML forms.

7.2.3 VERSIONING

As applications change, new versions are made and released. These new versions may have new conceptual models. So first and foremost, an application must somehow mark the objects that it produces to indicate which version of the application produced them. The concepts associated with later versions of the application and their conceptual models are often missing from earlier versions, causing forward incompatibility.

As new versions become available, people may decide to start using them. Since people rarely all upgrade at once, various versions of applications are often in use concurrently. Also, users share their work, and expect their results to work together. This requires that applications and their conceptual models take into account the fact that objects that are imported may come from older versions of an application and reflect an older conceptual model. When this happens, not only must the implementations be converted, but so must the concepts. For example, when Microsoft Office moved to XML-based documents, some concepts changed, e.g., the whole model of embedded graphics in documents. Old documents had "graphics frames" (aka "canvasses") in which graphical objects could be drawn, and graphical objects could not be drawn outside of graphics frames. New documents have graphical objects that can be overlaid on anything, and the graphics frames are gone. A user who understood the old model may not understand the new one.

What happens when an object being imported was created by someone using a *newer* version of the application with a newer conceptual model? That is, the application is trying to handle something that comes from its future. In this situation, some applications crash (bad); some refuse to proceed (not great); some strip out what they do not understand (better); some preserve and protect what they do not understand and pass it back out undamaged to others to use (good). In addition an application can appeal to users for help. For example, in a calendaring application, when in a new version meetings are differentiated by whether participants are in different timezones, users had to be asked for help in marking meetings. existing in the older version for which that information had not been required.

Supporting data-interchange between different versions of applications and their conceptual models requires careful thought and planning. It must be addressed in the conceptual model. Specifically, operations are needed to make the required conversions. The newer conceptual models may need to include descriptions from the older ones, and versions of the concepts need to be named so as to remove ambiguity. One solution is to tag concept labels with version numbers. This is not a solved problem.

7.2.4 UNANTICIPATED GROWTH

Despite all efforts to analyze the work in a task-domain, it is not possible to guarantee that the concepts in an application will align well with the actual situations users encounter. Sometimes this results from a lack of resources to see or consider every case in the domain. More often, the world changes and the design does not reflect the shift.

For example, many years ago, a photocopier was on an ocean-going barge. When the customer tried to order supplies, the order-entry clerk asked for an address to send them to, but the customer could not supply a normal address — only a list of port-of-call when the barge would be there. However, the form required a conventional street address, so the order could not be placed.

When the world (the task) and the application (the tool) cannot be made to align, what is the poor user to do? Beyond that, what are designers supposed to do to address user needs in such cases? How do we design for the unanticipated? How should the conceptual model reflect this inevitable result of working in a dynamic world? Here are some possibilities:

- *Backchannel*: Make it possible for users to tell the designer about the difficulty. While this will not solve the immediate problem, it may help in getting a fixed version released. For example, when applications fail (crash!), the implementations often offer to send a message to the implementers providing the technical details of the failure. Similarly, when the design fails, some applications support the user in sending a message to the designer giving the task circumstances. This "backchannel" is the means of getting a message "back" to the designer. To do this, the conceptual model must include concepts for communicating, addresses for designers, descriptions of task circumstances (e.g., open-ended text, photos).

- *Tailoring*: Make it possible for users to fix the application: to modify it to be as they would like it to be. This throws users into the design process, continuing the design in use. It is fraught with difficulty, because it requires that users be able to not only understand and change the design, but also be able to implement it. The impact on the conceptual model is to include all the concepts of designing and implementing. (Extensibility, discussed above, is a special case.)

  Even if all that is possible, the user may not want to switch to doing the application developer's work while in the middle of working on a task. Also, the timescale is probably wrong, since development work usually takes much longer than application usage. For example, when in the midst of writing something using a text-editing application, if a paragraph style has the wrong "next paragraph" setting, a user is not likely to edit the style at the expense of losing their writing flow.

- *Appropriation*: Find someone who has seen this problem before and has solved it, and appropriate their solution. This is a great approach, as it leads toward workable, even working, solutions. However, it requires that people be able to characterize their problem, know enough about other people's activity to know about earlier difficulties, find them, and get their help. Again, it usually cannot be done quickly enough to suit the circumstances. To support this approach, add **users** and completed-**tasks** to the conceptual model.

- *Work-arounds*: Augment the objects and operations of the application with some user-supplied objects and operations to create a solution that is partly *inside* the application and partly outside. In practice, this is the route people usually follow. For example, the Xerox order-entry clerk (see above) handled the "no available mailing address" problem by writing a telephone number in the address field of the form, with the additional instruction "Call Bob". Business forms (the application) have *margins* on them; these are the normal place for everything in the task that doesn't fit the structure of the form (the conceptual model of the application). To enable this solution, the conceptual model is augmented with open-ended ways for recording **exceptions** of one type or another (e.g., illegal values in fields, additional information, challenges to the assumptions of the application), together with operations for setting them and searching them.

- *Special handling*: Provide a special way to deal with exceptional cases. This is a mindset that covers all of the above solutions, and is most relevant in situations where currently the only way of getting work done is through the application (e.g., all hiring must be done through the HR++ application). For example, InterMountain Healthcare requires that doctors treat patients either by following the established protocol (the usual case) or by doing what they deem to be better and document it (the special case). The two "lanes" are designed together: the protocol can be very clear (to achieve coherence); and the protocol committee meets every day (to respond to new things). Adding special handling to the conceptual model for an application requires concepts like *type of case* (usual, special), *special handling report*, and *status* (for tracking that everything in the end gets addressed). It also requires that both lanes be designed, and designed to work together.

- *User-created conceptual model and design*: Ideally, of course, it would be great if a person could engage and modify the application at the level of the conceptual model. (This can be seen as the most extreme example of "Tailoring" above.) They would open up the application, access the conceptual model which would be separately expressed, modify the conceptual model to fit their new needs, and let the application adjust everything to make the application reflect the modified conceptual model. This automatic adjustment might require some help from the user on how they wanted the changes to be exposed by the user interface; and it might require some help with aspects of the implementation (e.g., how much file space is the user willing to devote to storing the new objects added).

  As of this writing, concept-based application construction remains a research topic. User Interface Management Systems (UIMS) focused on generating user interfaces. Content

Management Systems and XForms cover forms-based applications. Model-based architectures, including UML, address object-oriented architectures. Missing are modeling languages that can adequately express the semantics of the conceptual models, user interface generation formalisms and mechanisms that take into account the technologies which deliver both the application (e.g., often now running on multiple servers) and the user interface (e.g., often running on a phone, tablet or other mobile device).

We therefore encourage the research community to revisit the (semi-)automatic generation of applications from conceptual models.

7.2.5 EMBEDDING IN SOCIAL DOMAINS

Evolution of software applications is rarely an individual matter. Most task domains are collaborative; people's changes can affect others. Each person responds to their own view of the situation. Soon the work can be riding off at high speed in many directions.

Therefore, changes must be considered in the context of the collaborative (cooperative and competing) activity of the task domain (or domains, since activity often cross-cuts and interactions). People negotiate, contest, support, and struggle. Most task-domains are social domains; the social activity encompasses the activity of the tasks and the applications.

As with the methods of anticipating the unanticipated (see *work-arounds* and *special-handling*, above), applications can be designed to also support the social aspects of the activity that encompasses the task activity they are built to support. Of course that has implications for their conceptual models.

For example, the intense activity that accompanies closing an organization's books at the end of the quarter or the fiscal year is a highly negotiated set of conversations. Financial applications address the anticipated final financial decisions; they rarely address — indeed, they often deny the existence of — the conversations that lead to those decisions. Therefore, every manager in every business invents their own way of supporting these conversations. It might help to extend the conceptual model of financial programs to support all those sticky-notes and e-mail, and to support the trading that goes on between managers in order to really do the work well.

Perhaps support for conversations, and the concepts that support them, should be a critical part of all applications. Evolution may not be so optional after all.

CHAPTER 8

Process

For any real-world application that embodies more than a handful of concepts, conceptual design can be expected to take a development team at least two weeks of meetings, drafts, discussions, and revisions. This chapter describes how to do this most efficiently and effectively.

8.1 FIRST STEP OF DESIGN

Software design and development, when properly centered on the intended users and tasks, includes many different types of work:

- gathering and prioritizing requirements (aka understanding user needs)
- design (all types)
- evaluation (informal and formal)
- implementation (prototype and product/production)
- documentation (internal and external, online and printed)
- release
- support

Although the activities are listed here in a logical order, the listed order does *not* represent the order in which these activities actually occur; elements of *all* of these activities occur and recur through development, regardless of whether that is planned.

Building a conceptual model is the foundational, and therefore usually earliest, step in the process that is considered part of *design* (in contrast to requirements gathering).

8.2 START WITH USER RESEARCH

User research provides hints about objects, operations, and attributes.

An initial breakdown of the objects, operations, and attributes in a task domain can often be obtained from transcripts of worker interviews and contextual observation sessions. When people describe to researchers how they work, or when they converse with others while working, they use nouns, verbs, and adjectives. Nouns usually refer to *objects* (e.g., "bank account", "paragraph", "photo"), verbs usually refer to *operations* (e.g., "open", "delete", "reconcile"), and adjectives usually refer to *attribute values* (e.g., "red", "loud", "bright"). Thus, transcripts from user research can provide a first draft of an objects/operations analysis.

The first draft often must be adjusted, because operations are sometimes expressed as nouns (e.g., "... then I send out *invites* for the meeting") and objects are sometimes expressed as verbs (e.g., "I find cheap hotels in a city by *googling* them"). Furthermore, although attribute *values* are usually expressed as adjectives, attribute *names* are usually expressed as nouns (e.g., "Next I set the *interest rate* of the loan"), making it necessary to decide which nouns in an interview transcript are objects and which are attributes.

Nonetheless, using the nouns, verbs, and attributes from user research transcripts is a useful first step.

8.3 THE CONCEPTUAL MODEL NEEDS A PLACE AT THE PROJECT TABLE

Figure 8.1 shows the activities that make up the development process. It is drawn as a cycle, because

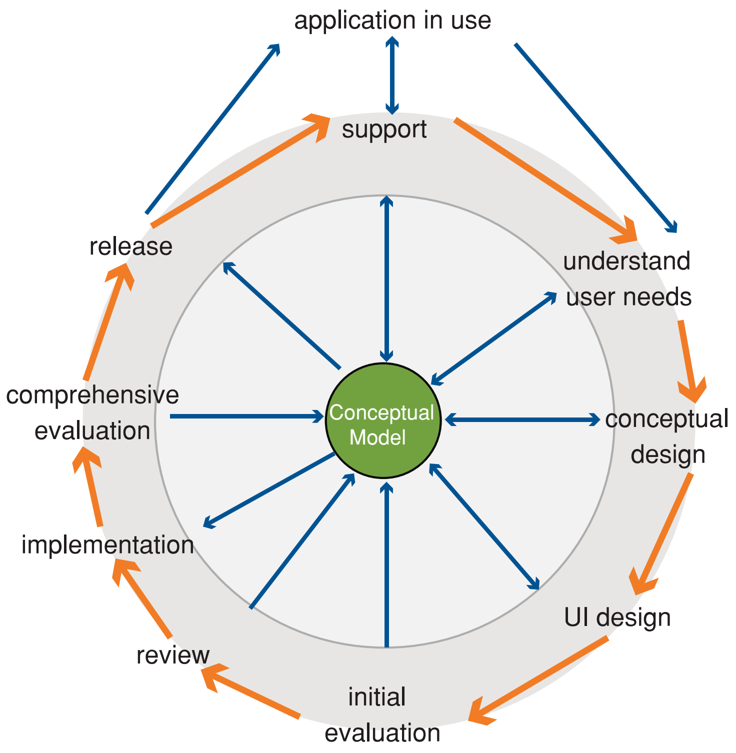


Figure 8.1: Conceptual design's place in a user/task-centered design process.

release leads to the application in use, with support supporting, which leads to better understand of user needs, and the beginning of the next round. Often, a number of versions are under consideration at once, so choices can be made as to where to apply learning, improvements and extensions. Sometimes this cycle is ponderous; sometimes it is agile. Sometimes these activities are sequential; but more often they are concurrent.

Each activity in this on-going cycle has its own issues that it must confront and resolve: support has to try to keep users happy, implementation has to create robust software, UI design has to ensure usability, conceptual design has to ensure usefulness for supporting user tasks, release has to ensure impact through well-timed delivery, and so on.

8.3.1 COORDINATION IS REQUIRED

All of these pressures on development must be coordinated. When realities conflict, they must be negotiated to produce a shipping application. Often these negotiations can produce novel solutions where everyone can win. New solutions may require a number of activities to change to incorporate it.

The practices for this negotiation vary widely across projects of different sizes, and different theories of development management. They are formal and informal, may involve documents or talk over lunch, may move between formally adjacent activities, or may shortcut the formal structure entirely.

Whatever development process is used, it is valuable to recognize that this negotiation must take place. A good development process therefore provides a place and occasions for this negotiation to happen. Metaphorically, this can be seen as a table around which all the activities sit and discuss their realities, negotiate, and create solutions.

Our view is that agreement on, and changes in, the conceptual model are crucial subject matters for those representing the activities gathered at this development table.

8.3.2 ONE TEAM-MEMBER SHOULD DRIVE THE CONCEPTUAL DESIGN

The entire team should have input into the conceptual model, but designing by committee can waste a lot of time. One person — probably the product architect or the lead UI designer — should be in charge of driving the objects/operations analysis. Based on whatever prior information they have (e.g., marketing research results, user research results, task analysis) they may want to draft a first-draft object/operations analysis and present it in a team-meeting for feedback.

Alternatively, the person in charge of the conceptual model may prefer to ensure team-members' buy-in by holding a team brainstorming session to produce a first-draft analysis. However, the process is started, the best way to proceed is by producing successive drafts and presenting them to the team for feedback.

8.3.3 INCLUDE DEVELOPERS, BUT KEEP THE CONCEPTUAL MODEL FOCUSED ON TASKS

Because of the similarity between the object-oriented software design and the object/operations analysis that is part of building a conceptual model, it helps to have OO programmers involved. They understand, for example, what it means to attach operations and attributes to objects, and how objects can be related to other objects. They can help inexperienced team members decide what is an object, an operation, an attribute, and how the concepts are related.

However, software developers should not be the ultimate deciders of what concepts are in the conceptual model, because they often want to expose concepts that should not be exposed: those that reflect the implementation rather than users' tasks.

8.4 USE THE CONCEPTUAL MODEL TO COORDINATE DEVELOPMENT

The conceptual model should be a central coordination point for development team members and their respective contributions: almost everyone on the team should orient towards the conceptual model as they design and develop the application.

The centrality of the conceptual model and its potential role in orchestrating the design process has one very strong implication for design activities and their relationship with the conceptual model:

Changes to the conceptual model are *team* decisions. Unilateral addition of concepts to the conceptual model by any team member is not allowed.

For example, if a programmer thinks a new concept needs to be added to the software and exposed to users, she must first persuade the team to add the concept to the conceptual model; only then can it appear in the software. Similarly, if a documenter discovers the need to introduce an additional concept to explain how to use the application, that change must first be reflected in the conceptual model, with the team's buy-in; then it may appear in the documentation.

When a change is made unilaterally, it can seem relatively inexpensive to the person proposing it. However, when all the impacts of a proposed design change are revealed by considering their impact on the conceptual model and the resulting impact on everyone else, the full cost can be considered before committing to the change or deciding against it.

8.5 REPRESENTING CONCEPTUAL MODELS

A conceptual model can be represented in many different ways. It can be represented in the form of an outline consisting of object-types and sub-types, each with operations and attributes (see Chapter 5). It can be represented in a table or spreadsheet, with objects in rows and actions and attributes in columns. It can be encoded in a computer-interpretable modeling language such as Unified Modeling

Language (UML). It can also be represented as one or more diagrams, such as a class diagram (see Fig. 8.2), an object-relationship diagram (see Fig. 8.3), or a concept map [Dubberly, H., 1999].

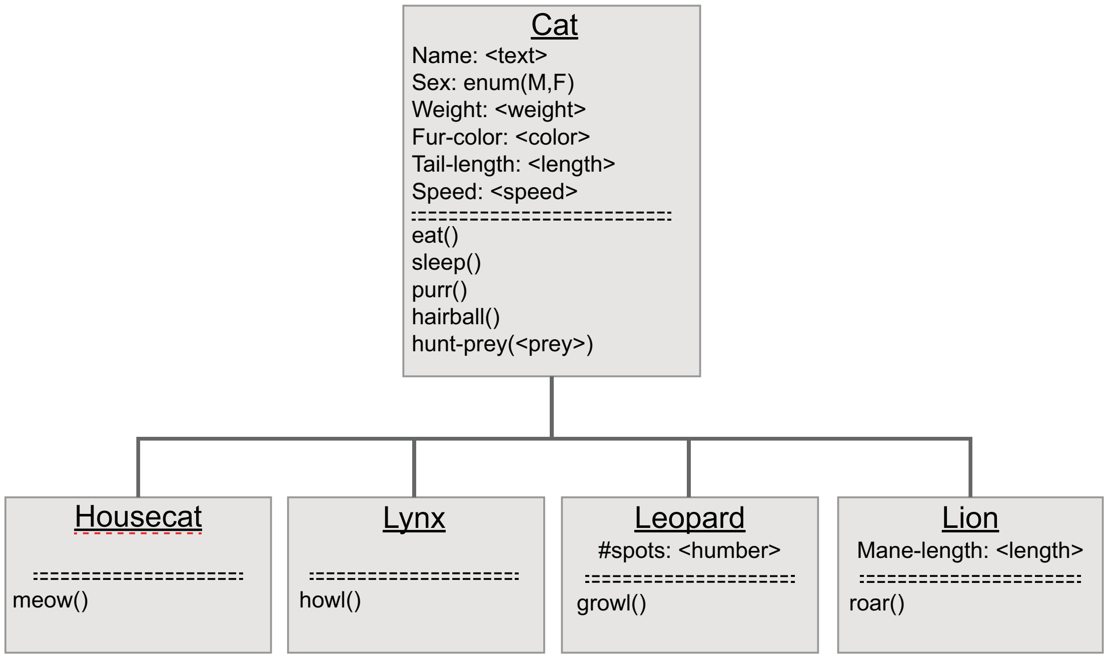


Figure 8.2: Class diagram for types of cats, with attributes and operations.

Tools are available that convert UML code into application source code, so representing the model in UML facilitates the development of application code underlying the conceptual model. However, it may *hinder* understanding of the model by non-programmer members of a development organization. It can also add pressure to include implementation concepts in the conceptual model so they will be automatically generated, which could damage the user-task focus of the conceptual model.

In our experience, when the model is shown to team-members to get their feedback, it usually suffices to represent it as an outline, a spreadsheet, or an object-relationship diagram.

8.6 ITERATE, ITERATE, ITERATE

People new to the idea of conceptual models often see them as an extra step in a linear, "waterfall"-like development process, e.g., determine requirements, ***develop conceptual model***, design user interface, code user interface and back-end software, test, release. In such a linear process, once each step is complete, its products, documents, code, etc. — are final and never revisited or revised, and later steps are based strictly on earlier ones.

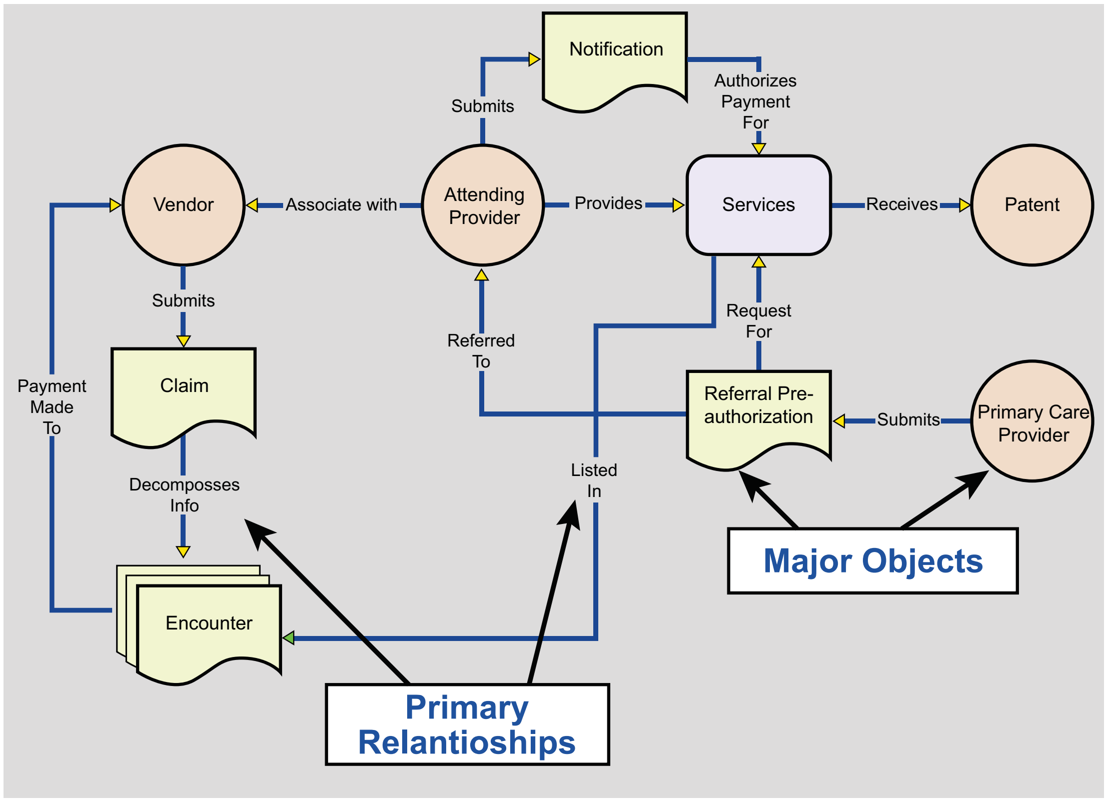


Figure 8.3: (*Courtesy of Jon Meads, copyright © 2011 Jon Meads.*)

This view is at the root of many current-day software managers' resistance to developing a conceptual model. Today's software managers often prefer to follow a development process that includes iteration and refinement, and that does not rely on big up-front design efforts and specification documents. This approach is based upon the idea that the initial conceptual model won't be completely right, so a user interface based on it will also be wrong.

It is true that first attempts at conceptual models are often not quite right. It is tough to get it right the first — or even the fifth — time. However, it is *not* true that developing a conceptual model rules out iterative design and development.

In fact, the process is *rarely* linear — even when organizations *try* to follow a "waterfall" development process. As design proceeds from conceptual model to user interface to implementation, it is likely that downstream design and implementation, not to mention user-testing, will expose problems in the conceptual model, indicating the need for changes.

In Fig. 8.1 (above), feedback loops through the conceptual model provide places where new understanding and evaluation findings can cause a return to a prior stage to revisit and revise that stage's output. Early usability testing can, and should, be designed to accelerate this process: low fidelity, quick prototypes can be focused on the important parts of, and questions in, the conceptual model. Lightweight usability testing can thus evaluate the conceptual model as well as the UI design.

In particular, a conceptual model, once designed, can be tested on users or other task-domain experts before further design work is done. Such testing may expose conceptual "holes" that can be corrected by revising the model, or it may expose new requirements, thereby necessitating a return to the user-needs (and prioritization) stage. Even after developers decide that the conceptual model is stable enough to proceed, later design and evaluation work may expose problems in the conceptual model, requiring revisions. Indeed, gaps, problems, and unnecessary concepts in a conceptual model may be discovered in pre-release testing or even after release.

If testing exposes problems in the conceptual model, developers should go back and change it. They should resist the temptation to treat the conceptual model as "dead" after an initial UI has been designed from it. If developers don't keep the conceptual model current as they "improve" the UI design, they will regret it in the end, when they have no single coherent high-level description on which to base the user interface, the user documentation, training, or later system enhancements.

Of course, changing the conceptual model can be painful: it affects the user interface, the documentation (in all its languages), and possibly even the implementation. The entire team is affected. But the conceptual model is the single most important part of your design. Therefore, it pays to make it as simple and task-oriented as possible, then do whatever you need to do to reconcile the rest of the design with it. Otherwise, the application's users will have little chance of understanding the user interface, because it will be based on a muddled conceptual model.

8.7 INCLUDING CONCEPTUAL MODELS IN AGILE DEVELOPMENT

An increasing number of software development organizations use Agile methods or methods similar to Agile. Agile development involves, among other things, rapid development and testing of successively more functional versions of the intended application [Ambler, S., 2005]. Testing in Agile development makes use of team-members who are task-domain experts, i.e., user-representatives[8]. They provide feedback on the design during weekly or bi-weekly team "scrum" meetings. Their role is to ensure that all critical requirements are met and to keep the design focused on the tasks [Beck and Andres, 2004].

Some might argue that designing conceptual models does not fit into Agile development, due to another characteristic of Agile development: its avoidance of large up-front design efforts and specification documents. Agile teams tend to prefer to specify a design in small pieces, on cards that each specify a single function or control. The idea is that the requirements and the overall UI design cannot be successfully predicted and specified, but rather will emerge as development proceeds.

[8]Sometimes these are actual users and sometimes they are proxies for users, e.g., developers familiar with the target task-domain from having formerly done that job.

The argument that conceptual models don't fit into Agile development can be countered in two ways. First, Agile and related methods were initially developed by software engineering experts, not interaction design experts, so early writings on Agile had blind spots concerning interaction design, usability, user-centered design, and how they fit into the methods. According to Bangston, A. [2001]:

> XP tends to promote a very tight focus; don't worry about what's coming, just code the card you're holding. … Unfortunately, … this approach can lead to disjointed, awkward or unnecessarily complicated interfaces designed around back-end functionality rather than the user's end goals. Designing an efficient and elegant user interface requires some conception of what steps comprise a given task, and how tasks interrelate to create an application's flow.

In other words, design-as-you-go methods might work for developing device drivers, operating system kernels, compilers, or other software that has little or no user interface, but design-as-you-go does not work for applications with significant user-interfaces, such as airline reservation websites, air-traffic control systems, or even flight-simulation games.

In recent years, user experience experts and Agile/XP experts have attempted to "marry" their methods. One result is that they acknowledge the need for some up-front design. For example:

- Scott Ambler, an Agile/XP guru, says that the overall architecture of an application should be modeled in a "phase 0" before the normal code-test-revise cycles start [Ambler, S., 2005]. That "phase 0" model would include a conceptual model.

- Larry Constantine, a UI consultant, says "some minimum up-front design is needed for the UI to be well-organized and to present users with a consistent and comprehensible interface." He and his colleague Carolyn Lockwood developed a streamlined "usage-centered" design methodology that they find fits well into Agile/XP development [Constantine, L., 2002].

- Jon Meads, a UI consultant, finds that user/task-centered design fits with Agile methods if the design (including iterations of it) occurs mainly in the Inception and Elaboration phases of a project, while the Agile code-test-revise cycles occur in the Construction phase (Fig. 8.4). One way to regard this is that iterative refinement of the conceptual design is no different than iterative refinement of the implementation code. Both are necessary.

Second, as described above, it is simply false that conceptual design only fits into a linear "waterfall" development process. User/task-centered design, of which conceptual design is one part, shares Agile's rejection of the "waterfall" model. It recognizes that customer requirements are understood better over time, and change even after they are understood. It acknowledges that designs evolve, but advises against beginning implementation or even detailed user-interface design until you have at least a preliminary version of a task-focused conceptual model.

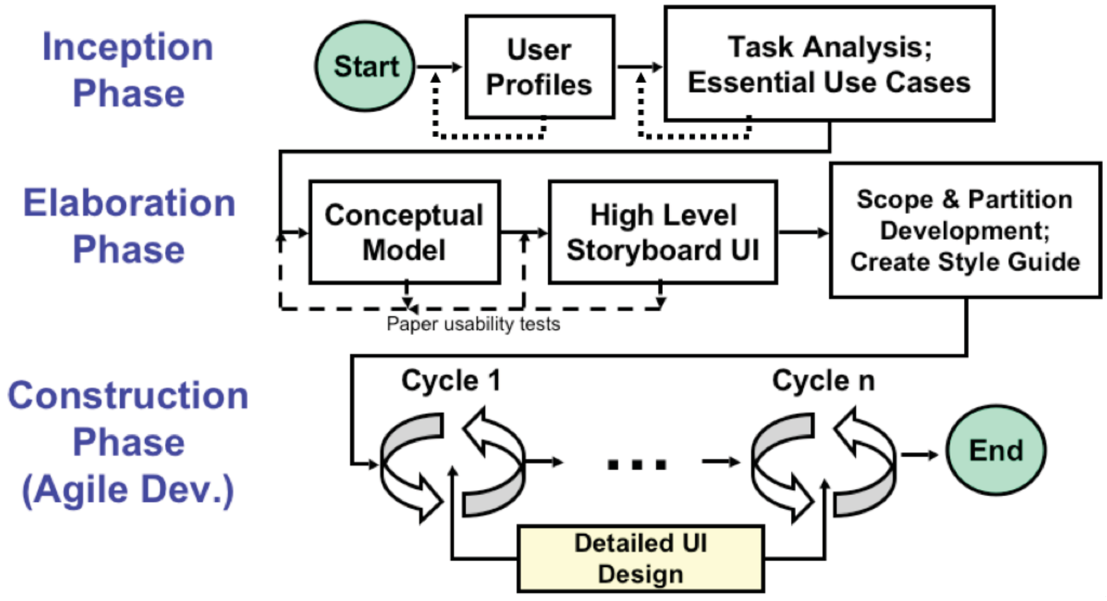


Figure 8.4: How Conceptual Models fit into Agile development (*Adapted from Jon Meads).*

On the other hand, the conceptual model, once it exists, should not be considered carved in stone. Quite the opposite, it should be tested on users and revised and improved *before* developers invest time, money, and egos on implementation code. After implementation starts, the testing and revision continue, and the design continues to evolve.

8.8 TESTING CONCEPTUAL MODELS

When designing a software application, regardless of whether the development team is following an Agile process or a more conventional one, an important rule to follow is:

Test early and often.

Designers should accelerate the discovery of problems in the conceptual model by actively seeking feedback from users or other task-domain experts as early in the process as possible, before the user interface is completely designed or even begun. Focus groups can consider the concepts of the conceptual model — objects, attributes, operations,, and relationships, with their terminology — and the task-flow that it implies more-or-less directly. Low fidelity, quick prototypes can be used to test important parts of, and issues in, the conceptual model.

Thus, lightweight usability testing can evaluate the conceptual model as well as the UI design. In general, the earlier developers test and evaluate their conceptual model, the less likely it is that conceptual design problems will surface late in development or after release.

CHAPTER 9

Value

Software development managers often balk at including a conceptual design phase in development. "How long will that take? Won't it slow us down? I don't like the idea of spending two or three weeks developing a conceptual model — let's just start designing the screens so our implementors can get coding ASAP!"

Developing a task-focused conceptual model for an application that all major stakeholders agree on does indeed take a few weeks. However, that investment usually pays off handsomely by clarifying, focusing, and speeding later development steps. In other words, developing a conceptual model is not simply an additional cost for a project; it produces outputs that are useful or even necessary for later steps, and that therefore can *save* development time and cost.

In this chapter, we list the downstream benefits of developing a conceptual model at the start of the design process.

9.1 PRODUCES A VOCABULARY

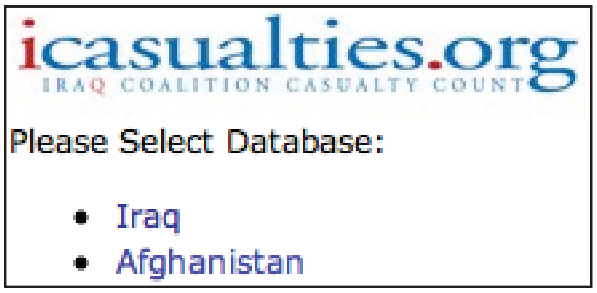
Once the development team assigns names to the conceptual objects, operations, and attributes that the application will expose to users, they have a *vocabulary* of terms (see Chapter 4) to be used in the application and its documentation. As the user interface is developed, the software coded, and the documentation written, the vocabulary can be consulted to ensure that terms are used consistently throughout.

The entire team develops the vocabulary, but it is best managed and enforced by a single person, such as the product manager, the product architect, or the team's lead technical writer. Whoever gets the vocabulary-manager job should keep the vocabulary up-to-date as concepts and terms change. The vocabulary-manager should also constantly be on the lookout for inconsistencies in what things are called in the application and its user documentation. For example:

> Hey, Anoop, it's Sergei. Got a minute? On your pages in our customer-service website, you use the term "bug report". But our agreed-upon term is "service request", remember? That's what's in the vocabulary. Where's the vocabulary? At the project's wiki. Can you please change "bug report" to "service request" on all your pages? We're running usability tests on Thursday, so I'm hoping you can make these changes by Wednesday. You will? Great, thanks.

Applications developed without a vocabulary often exhibit one or both of two common user interface "bloopers" [Johnson, J., 2007]:

1. ***non-task-relevant terms:*** terms for concepts not in the conceptual model, i.e., that shouldn't be exposed to users (see Fig. 9.1 and Sidebar 4),

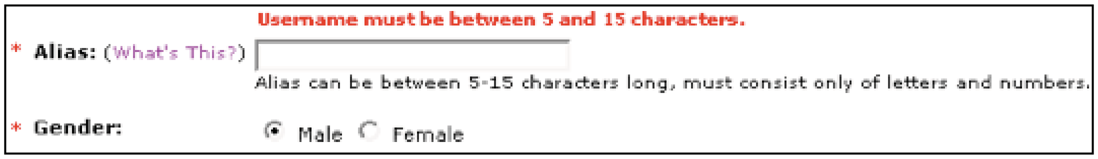


Please Select Database:

- Iraq
- Afghanistan

Figure 9.1: iCasualties.org asks users to select a "Database," which is an implementation concept, not a task-related concept. "War" or "War-Zone" would be more task-relevant.

2. ***inconsistent terminology:*** which takes two forms: a) multiple terms for a single concept (see Fig. 9.2), and b) the same term for multiple distinct concepts, i.e., overloaded terms (see Fig. 9.3).



Username must be between 5 and 15 characters.

* Alias: (What's This?)

Alias can be between 5-15 characters long, must consist only of letters and numbers.

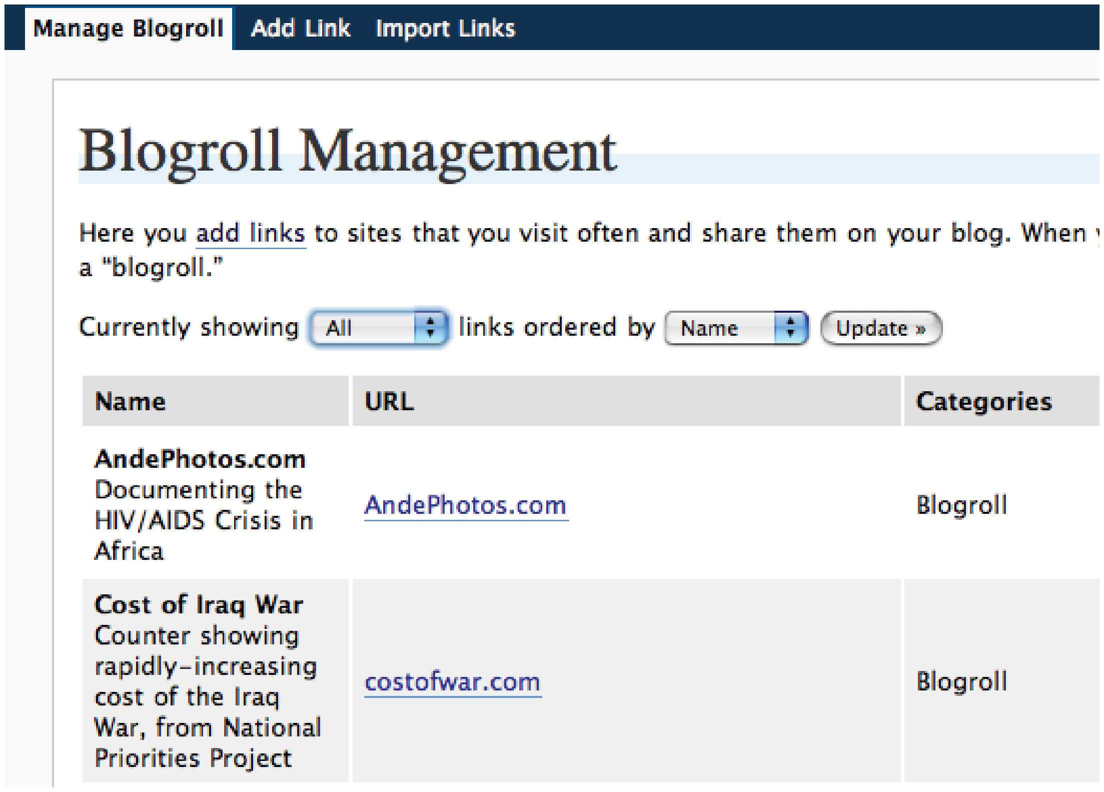
* Gender: [✓ Male] Female

Figure 9.2: eVite.com labels the first field "Alias" but the error message calls it "Username."

To ensure that all terms that users encounter are relevant to their tasks, only terms that correspond to concepts in the conceptual model should be used in an application and its documentation. Terminology that is not relevant to users' tasks should be kept entirely out of software applications and their documentation.

Furthermore, the terminology for concepts should be extremely consistent; otherwise users — especially new ones — will be confused and will take longer to become proficient in using the application. To ensure consistency, the design rule that application designers should follow is:

Same name, same thing; different name, different thing.
(Jarrett, www.formsthatwork.com)



Blogroll Management

Here you add links to sites that you visit often and share them on your blog. When
a "blogroll."

Currently showing All links ordered by Name Update »

Name	URL	Categories
AndePhotos.com Documenting the HIV/AIDS Crisis in Africa	AndePhotos.com	Blogroll
Cost of Iraq War Counter showing rapidly-increasing cost of the Iraq War, from National Priorities Project	costofwar.com	Blogroll

Figure 9.3: At WordPress.com, "Blogroll" means both a list of external links and a category setting on the list.

To keep foreign concepts and terms out of the user interface, the vocabulary-manager should check for concepts and terms in the interface, software or documentation that aren't in the conceptual model or the vocabulary, and resist them. For example:

> Hey Sue, this screen refers to a "hyper-connector", which isn't in our conceptual model or vocabulary. Is it just the wrong name for something we already have in our conceptual model, or is it something new? If it's something new, can we get rid of it?

In these ways, designing a conceptual model for an application facilitates a development team's creation and use of a product vocabulary, which in turn results in an application that is easier for people to learn and understand.

Sidebar 4: Application Using Non-Task-Related Terminology

A company developed a stock-investment application for stockbrokers to use when placing investments for their clients. The application let stockbrokers create and save templates for frequent transactions. It gave them the option of creating templates purely for their own use, or to be used by their co-workers. Templates shared with other users were stored on a server; private templates were stored on the stockbroker's office PC. The application referred to the two storage options as follows:

- To create a shared template: Create Database Template
- To create a private template: Create Local Template

The developers used "database" for shared templates because shared templates were stored in a database. The developers used "local" for private templates on the users' own PC because that's what "local" meant to them. More task-relevant terminology would be:

- To create a shared template: Create Shared Template
- To create a private template: Create Private Template

9.2 FACILITATES CREATION OF HIGH-LEVEL TASK SCENARIOS

Once a conceptual model has been crafted for an application, the designers can write scenarios depicting people using the application to perform specific tasks, using only concepts from the conceptual model and terminology from the vocabulary. Such scenarios are sometimes called *use-cases*.

Conceptual-level task scenarios are useful in checking the soundness of the design, e.g., in product functional reviews. Secondly, they can be presented directly to users in product documentation and training. Thirdly, because conceptual task scenarios describe tasks and goals without revealing the keystroke-level user actions required to achieve those goals, they can be used as task descriptions in usability tests. Finally, they provide the basis for later, more detailed scenarios written in the terminology of the eventual user interface design.

In the case of the bank account application, for example, it should be possible to write scenarios such as this:

> John checks his account balance. He then deposits a check into his checking account and transfers funds there into his savings account.

Note that this scenario refers to task-domain and conceptual model objects and actions only, not to details of any user interface. The scenario does not indicate whether John is interacting with a GUI on a personal computer or a voice-controlled interface over a telephone. However, it does specify functionality that the application must have, as well as placing some constraints on the design. For a more comprehensive conceptual task scenario, for a photo management application, see Sidebar 5.

Sidebar 5: Conceptual Task Scenario

The following is a task-scenario for a photo management application, expressed at a conceptual level, without reference to a specific user interface:

> Sally just returned from a two week vacation in San Francisco, and wants to show her photos to her friends. She's already deleted the truly bad photos from her camera, but now she needs to download the remaining ones so she can see them in larger size and edit them down to a slideshow of the best shots. She connects her camera to her computer, starts the photo management application, and downloads the photos from the camera. After the photos download, she browses through them. Some that looked OK on her camera's small screen are clearly out of focus, so she deletes them. A few shots of people have the red-eye problem, so she fixes that. She creates an album and names it "SF 2012", and puts the best 50 photos into it. She moves some photos around in the show so it makes more sense, then sets the slideshow to fade between photos. She views the slideshow to make sure it flows well, then quits the application.

This scenario says almost nothing about the photo management application's user interface — only that it runs on a personal computer — but says a lot about the application's functionality.

9.3 FACILITATES CREATION OF USER DOCUMENTATION, TRAINING, AND SUPPORT

As mentioned above, vocabulary and task-scenarios that come out of the conceptual model are important inputs to the writing of user documentation and training.

However, more importantly, a conceptual model provides those who write user documentation, training and support materials with a high-level, conceptual view of the application. It gives them a "big picture" view of the application, which helps them write instructions for using it that make sense to users.

Furthermore, the vocabulary of the conceptual model provides a clear point of contact for people who localize the application to particular contexts. The most common such localization is translating the application into other languages. Less common, but at least as important, is creating specialized versions of the application for targeted situations. For example, corporations may have internal vocabularies which when applied make the application talk the local lingo (e.g., in a law firm, **folders** may be called "redwelds"; in a shipping company, crates may be called "loads"). Having a single point of contact can greatly decrease the dependence of specialization activities on tracking details of the user interface design.

In addition, an application's conceptual model can be presented directly to users in its documentation, training, and support materials, thereby giving users a jump-start on forming an effective mental model.

9.4 FOCUSES THE USER INTERFACE DESIGN: GIVES DESIGNERS A CLEAR TARGET

The *user interface design* (or, if you prefer, interaction design) translates the abstract concepts of the underlying conceptual model into concrete presentations, controls, user-actions, and task-flows. It does that *regardless* of whether or not the conceptual model was explicitly and carefully designed beforehand to make sense. In other words, the user interface is a projection of *some* conceptual model, whether that conceptual model was intended or not and whether it is coherent or not. Therefore, if designers want their application to have a coherent user interface, they should explicitly design a conceptual model for it.

An explicitly created conceptual model for an application can be regarded as a *declaration* of concepts that the application will (and will not) expose to users. It can be treated like a contract: the application's user interface can present a concept (object, operation, attribute, relationship) to users if and only if the concept is in the conceptual model.

A conceptual model gives designers a clear target for what the user interface must deliver to users. The presentation and keystroke-level interaction of the objects, operations, attributes, and relationships must be designed. The conceptual model then offers the basis for tests of how well the user interface works, that is, whether users can manipulate the objects through their UI representations as the designers intended.

The conceptual model constrains the user interface: concepts that are not in the conceptual model should not be presented by the user interface. This focuses the interface design effort, not only keeping non-task-relevant, potentially confusing concepts out of the interface, but also saving designers time and effort.

Once a user interface is designed, the conceptual task-scenarios (see above) can be fleshed out with user-interface details, even down to the level of keystrokes. For example:

> John double-clicks on the icon for his account to open it. A separate window opens showing the current balance. He clicks in the blank entry field below the last recorded entry and types the name and amount of a check he recently received. ...

9.5 JUMP-STARTS AND FOCUSES THE IMPLEMENTATION

Readers who are programmers will have noticed the similarity between the objects/operations analysis described here and the object-oriented analysis that is a common early step in software engineering. One big difference is that objects/operations analysis is restricted to concepts that are exposed to users while object-oriented analysis includes all concepts that are part of the implementation.

Nonetheless, having performed an objects/operations analysis provides a development team with a subset of the objects, object operations, and object attributes that must be implemented. Therefore, after the conceptual model has stabilized and agreed upon, software developers can begin implementing the internal structure of the required objects, operations, and attributes without waiting for designers to design the detailed user interface.

Continuing with the example of a personal banking application: once the development team knows what types of accounts the application offers and what the operations and attributes of each type of account will be, programmers can begin coding those objects, without knowing yet how accounts will be displayed, edited, or manipulated. They need not wait for the user interface design.

Thus, contrary to the fears of many development managers, developing a conceptual model does not delay the start of user-interface coding. It may actually allow it to begin earlier.

9.6 SAVES TIME AND MONEY

A bonus benefit that results from the above-described benefits is that creating a conceptual model for an application can save developers time and money. Yes, creating a good conceptual model takes time — a week to a month depending on the complexity of the task-domain and the intended application.

However, by facilitating creation of documentation and training, guiding and constraining the user interface, jump-starting the implementation, and focusing the design process, a conceptual model often saves a development team a great deal of unnecessary effort, thereby *shortening* the rest of the development process by much more than the time it took to develop the conceptual model.

CHAPTER 10

Epilogue

A good Conceptual Model (CM) should be at the core of the design of every artifact that people use to help them get their work done.

This book argues that a conceptual model:

- helps focus the design of the application by coupling the design to the tasks that the user is doing;
- supports having a good process for developing that design into a product;
- makes using the application easier for the user.

As outlined in the Introduction, the book discusses the context of use of tools, and hence the target of good design. It describes what a conceptual model is, and how it is structured. It looks at recurrent issues — both essential and optional concerns in modeling, and concerns in design process. It argues that resources invested in building conceptual models repay that effort.

Having a focused conceptual model as a part of application development is not sufficient for designing a good tool for users. Good conceptual models must be joined by good work on user interfaces, implementation, documentation, testing, and support, all coupled by good design processes. All of these activities can meet and align through developing a good,conceptual model.

A clear, task-based, current, documented conceptual model can act as a meeting point for all those engaged in building and maintaining an application. As a result, users will have a more understandable application, and, more importantly, a more effective tool for getting their work done.

Bibliography

Abowd, G. and Dix, A. (1992) "Giving undo attention" Interact. Comput. 4, 3 (December 1992), 317–342. DOI: 10.1016/0953-5438(92)90021-7 Cited on page(s) 62

Ambler, S. (2005) "The Agile System Development Lifecycle", Ambysoft online article, `http://www.ambysoft.com/essays/agileLifecycle.html` Cited on page(s) 77, 78

Bangston, A. (2001) "Usability and User Interface Design in XP."Online article: `http://www.ccpace.com/Resources/documents/UsabilityinXP.pdf` Cited on page(s) 78

Beck, K. and Andres, C. (2004) *Extreme Programming Explained: Embrace Change, 2nd Ed.* Reading, MA: Addison Wesley. Cited on page(s) 77

Beyer, H. (2010) *User-Centered Agile Methods*, Morgan & Claypool.
DOI: 10.2200/S00286ED1V01Y201002HCI010 Cited on page(s)

Beyer, H. and Holtzblatt, K. (1997) *Contextual Design*. Morgan Kaufmann. Cited on page(s) 24, 29, 30

Bittner, K. and Spence, I. (2003) Use Case Modeling. Addison-Wesley. p. xvi. Cited on page(s) 20

Buxton, B. (2007) *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann. Cited on page(s) 23

Card, S. (1993) Discussant comments, at HCI Consortium, Boulder ,CO, Cited on page(s) 15

Card, S. (1996) "Pioneers and Settlers: Methods Used in Successful User Interface Design", in M. Rudisill, C. Lewis, P. Polson, T. McKay (eds.), *Human-Computer Interface Design: Success Cases, Emerging Methods*, Real-World Context, Morgan Kaufmann. Cited on page(s) 15, 30

Constantine, L. (2002) "Process Agility and Software Usability: Toward Lightweight Usage-Centered Design", *Information Age*, Aug-Sep. Cited on page(s) 78

Cooper, A. (2004) *The Inmates are Running the Asylum: Why High Tech Products Drive Us Crazy and How to Restore the Sanity*. Pierson/SAMS, Cited on page(s) 29

Dubberly, H. (1999) "Understanding Internet Search", Dubberly Design Office, `http://www.dubberly.com/concept-maps/understanding-internet-search.html` Cited on page(s) 75

Halasz, F. and Moran, T. (1982) "Analogy Considered Harmful", Proceedings of CHI 1982, pages 383–386. DOI: 10.1145/800049.801816 Cited on page(s) 22, 53

Isaacs, E. and Walendowski, A. (2001) *Designing from Both Sides of the Screen: How Designers and Engineers Can Collaborate to Build Cooperative Technology*, SAMS. Cited on page(s) 53

Johnson, J. (1985) "Calculator Functions on Bitmapped Computers", *SIGCHI Bulletin*, July 1985, pages 23–28. DOI: 10.1145/378965.378971 Cited on page(s) 21, 53

Johnson, J. (1987) "How Faithfully Should the Electronic Office Simulate the Real One", *SIGCHI Bulletin*, July 1987, pages 21–25. DOI: 10.1145/36111.36113 Cited on page(s) 22, 53

Johnson, J., Roberts, T., Verplank, W., Smith, D.C., Irby, C., Beard, M., and Mackey, K. (1989) "The Xerox Star: A Retrospective", *IEEE Computer*, September, 1989, pages 11–29. DOI: 10.1109/2.35211 Cited on page(s) 21, 30, 54

Johnson, J. and Henderson, D.A. (2002) "Conceptual Models: Begin by Designing What to Design", *Interactions*, Jan-Feb 2002, pages 25–32. DOI: 10.1145/503355.503366 Cited on page(s)

Johnson, J. (2007) *GUI Bloopers 2.0: Common User Interface Don'ts and Dos*, Morgan Kaufmann Publishers. Cited on page(s) 54, 81

Lombardi, V. (2008) "Concept Design Tools", Digital Web Magazine, Sept. 30.Web: `http://www.digital-web.com/articles/concept_design_tools/` Cited on page(s) 23

Moran, T.P. (1983) "Getting into a system: External-internal task mapping analysis", In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI '83)*, Ann Janda (Ed.). ACM, New York, NY, USA, 45–49. DOI: 10.1145/800045.801578 Cited on page(s) 7

Newman, W. and Sproul, R. (1973) *Principles of Interactive Computer Graphics*. New York: McGraw-Hill. Cited on page(s) 30

Norman, D.A. and Draper, S.W. (1986) *User Centered System Design: New Perspectives on Human-Computer Interaction*, Hillsdale, New Jersey: CRC. Cited on page(s) 24

Norman, D.A. (2010) *Living With Complexity*. MIT Press. Cited on page(s) 15

Ostrom, E. (1990) *Governing the Commons: The Evolution of Institutions for Collective Action*, Cambridge University Press. Cited on page(s)

Suchman, L.A. (2007) *Human-Machine Reconfigurations: Plans and Situated Actions*, Cambridge University Press. Cited on page(s)

Washizaki, H. and Fukazawa, Y.. (2002)"Dynamic hierarchical undo facility in a fine-grained component environment." in Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications (CRPIT '02). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 191–199. Cited on page(s) 63

Young, R.M. (1981) "The Machine Inside the Machine: Users' Models of Pocket Calculators." *International Journal of Man-Machine Studies* (1), pages 51–85. DOI: 10.1016/S0020-7373(81)80023-5 Cited on page(s) 7

Authors' Biographies

JEFF JOHNSON

Jeff Johnson is President and Principal Consultant at UI Wizards, Inc., a product usability consulting firm that offers UI design, usability reviews, usability testing, and training (`http://www.uiwizards.com`). He also is co-founder of Wiser Usability (`http://WiserUsability.com`), a consulting firm specializing in elder usability and accessibility. He has worked in the field of Human-Computer Interaction since 1978. After earning B.A. and Ph.D. degrees from Yale and Stanford Universities, he worked as a user-interface designer and implementer, engineer manager, usability tester, and researcher at Cromemco, Xerox, US West, Hewlett-Packard Labs, and Sun Microsystems. He has taught at Stanford University and Mills College, and in 2006 was an Erskine Teaching Fellow at the University of Canterbury in Christchurch New Zealand. He has published numerous articles and book chapters on a variety of topics in Human-Computer Interaction and the impact of technology on society. He frequently gives talks and tutorials at conferences and companies on usability and user-interface design. His previous books are: *GUI Bloopers: Don'ts and Dos for Software Developers and Web Designers* (2000), *Web Bloopers: 60 Common Design Mistakes and How to Avoid Them* (2003), *GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos* (2007), and *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules* (2010).

AUSTIN HENDERSON

Austin Henderson's 45-year career in Human-Computer Interaction includes user interface research and architecture at MIT's Lincoln Laboratory, Bolt Beranek and Newman, Xerox Research (both PARC and EuroPARC), Apple Computer, and Pitney Bowes, as well as strategic industrial design with Fitch and his own Rivendel Consulting & Design. Austin has built both commercial and research applications in many domains including manufacturing, programming languages, air traffic control, electronic mail (Hermes), user interface design tools (Trillium), workspace management (Rooms, Buttons), distributed collaboration (MediaSpace), and user-evolvable systems (Tailorable — "design continued in use," Pliant — "designing for the unanticipated" and "scalable conversations"). These applications, and their development with users, have grounded his analytical work, which has included the nature of computation-based socio-technical systems, the interaction of people with the technology in those systems, and the practices and tools of their development. The primary goal of his work has been to better meet user needs, both by improving system development to better anticipate those needs, and by broadening system capability to enable users themselves to better respond to unanticipated needs when they arise in a rich and changing world.