CHAPTER 9

Toward Unified Models in User-Centered and Object-Oriented Design

William Hudson

Abstract

Many members of the HCI community view user-centered design, with its focus on users and their tasks, as essential to the construction of usable user interfaces. However, UCD and object-oriented design continue to develop along separate paths, with very little common ground and substantially different activities and notations. The Unified Modeling Language (UML) has become the de facto language of object-oriented development, and an informal method has evolved around it. While parts of the UML notation have been embraced in user-centered methods, such as those in this volume, there has been no concerted effort to adapt user-centered design techniques to UML and vice versa. This chapter explores many of the issues involved in bringing user-centered design and UML closer together. It presents a survey of user-centered techniques employed by usability professionals, provides an overview of a number of commercially based user-centered methods, and discusses the application of UML notation to user-centered design. Also, since the informal UML method is use case driven and many user-centered design methods rely on scenarios, a unifying approach to use cases and scenarios is included.

9.1 Introduction

9.1.1 Why Bring User-Centered Design to UML?

A recent survey of software methods and techniques [Wieringa 1998] found that at least 19 object-oriented methods had been published in book form since 1988, and many more had been published in conference and journal papers. This situation led to a great

deal of division in the object-oriented community and caused numerous problems for anyone considering a move toward object technology.

The picture today could not be more different. In classes that I teach, and when I'm working with my development customers, UML is the predictable answer to any question concerning process or notation. A simple comparison of methods and processes[1] mentioned on the Web (see Figure 9.1) shows UML outstripping its nearest competitor (Object Modeling Technique, or OMT) by a factor of four. A leading online book retailer lists no fewer than 64 titles on the subject, with new volumes seeming to appear at the rate of two or three a month.

The focus on a single object-oriented notation has numerous benefits. Skills are more readily transferred between projects, communication between designers and developers

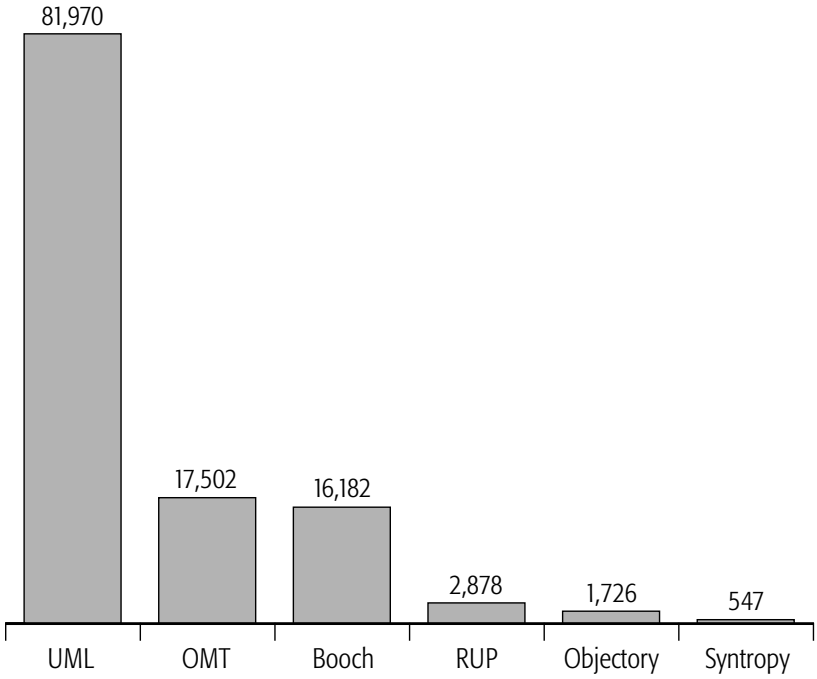


FIGURE 9.1 Web references to UML and object-oriented methods

Note: References were obtained by searching for each term (in English only) on *www.raging.com* in June 2000. Methods with commonly occurring names such as "OPEN" and "Fusion" could not be included for obvious reasons. "RUP" is the common abbreviation for the Rational Unified Process, which is discussed later in this chapter.

[1] There is quite a bit of disagreement over the meanings of "process" and "method" (compare Jacobson et al. [1992] and Olson and Moran [1995]). In this chapter, I have used the term "method" unless "process" has been used by the authors of the approach being discussed.

is easier, and development support in the form of software tools is steadily improving. Introducing user-centered techniques to UML would yield similar benefits, and also increase the awareness of developers to user-centered issues.[2]

This final point may seem both trivial and contentious. However, my own experience in developing interactive software and teaching for 30 years is that most systems are developed with little or no understanding of usability and user-centered issues. Raising awareness of these issues in a mainstream technology such as UML can only help to increase software usability.

9.1.2 Why Not Another New Method?

I argue against the introduction of another user-centered method for the following reasons.

- It perpetuates the object-oriented/user-centered divide. User-centered design needs to become a part of mainstream software development, not a collection of tributaries.
- Methods are not used in a formal way. Developers (object-oriented, user-centered or any other type) adopt techniques or notations that work for them. What most developers mean when they say that they are using a method is that they are using some parts of it. In the user-centered development survey presented later in this chapter, the most popular *technique* was used by 93.5 percent of respondents. In comparison, the most popular user-centered *method* was used by only 11.8 percent.[3]
- New methods are distractions to most development organizations. While some parts of the computer industry are perpetually searching for a new "magic bullet," most development managers live in dread of substantial technological change. Books, training, consultants, new staffing requirements, and reduced efficiency during the "learning curve" contribute to make this a stressful experience for all concerned.

Because UML includes use cases, it has the potential to be truly user-centered, given some appropriate adjustments. My premise is that introducing new or modified techniques to UML is, if nothing else, *psychologically* more acceptable to developers than introducing a new method. I also believe that this approach has a better chance of changing software development practice than the introduction of another new method.

---

[2] See Chapter 10 in this volume for a definition and discussion of user-centered design.

[3] Techniques are the constituents of methods and can be thought of as discrete high-level tasks for the user of a method.

9.1.3 How Can UML Be Made User-Centered?

A preliminary issue that I need to address is that UML is a language, not a method (hence UML, not UMM). Consequently, when developers state that they are using the UML method, they mean an informal method that has evolved around UML thanks largely to one of the earliest books on the subject: *UML Distilled* [Fowler 1997]. In addition, the UML notation draws heavily on the three *methods* that precede it: Booch [Booch 1994a], Objectory [Jacobson et al. 1992], and OMT [Rumbaugh et al. 1991]. The result is the informal method that I describe in Section 9.3.

Is UML a suitable basis for user-centered design? Like Objectory, the informal UML method is use case driven. It was Jacobson's original intention that use cases would allow systems to be built for their users [Jacobson 1987, p. 186]. However, it is now widely accepted [Cockburn 1997a, 1997b; Constantine 1994, 1995; Constantine and Lockwood 1999; Graham 1997] that use cases are too vague in their definition and varied in their use to be truly effective. Cockburn addressed this issue by trying to catalogue the variations and by focusing attention on a specific subset he called "goal-directed use cases." Constantine saw problems with unnecessary detail that led to premature design decisions and introduced a form of abstract use case he called "essential" (see Chapter 7). Meanwhile, Graham placed one foot firmly in the human-computer interaction (HCI) camp and suggested task scripts as an alternative (see [Graham et al. 1997]).

Not surprisingly, use cases top the list of problems that need to be addressed before the informal UML method can be considered user-centered. The complete list follows.

- *Confusion over use cases.* The purpose and content of use cases for user-centered design need to be refined and explained.
- *No separation of user and domain models.* Object-oriented methods in general do not acknowledge the difference between the problem domain and a user's understanding of the problem domain.[4]
- *No deliberate user interface design.* In many cases, design of the user interface is left to the developer who is responsible for the underlying functionality. This allows no proper opportunity to design the user interface as a whole.
- *Lack of contextual information.* User needs can vary dramatically according to context—that is, the set of circumstances and conditions under which a task is performed. In common with most other object-oriented approaches to software

[4] To be absolutely fair, it may be that some object-oriented methods intend the domain model to be a user's conceptual model, but few if any methods adequately explain what their domain models include.

development, UML does not take context into account, with the exception of brief task descriptions that may appear in use cases.

- *No usability evaluation.* This criticism is a little unfair but necessary. UML is primarily an analysis and design notation. It does not concern itself with how software is written and tested. However, as most usability practitioners will confirm, you can't have a user-centered approach without usability evaluation. (This is also the view taken by the ISO 13407 standard for human-centered systems design [ISO 1999].)

Usability evaluation is an example of one technique that we might incorporate into UML to make it more user centered. As there are dozens of techniques (see, for example, Hackos and Redish [1998] and Mayhew [1999]), how do we decide which are the most effective? After all, we can probably add only a small number to an existing method without appearing to hijack it.

The approach that I have taken is to conduct a survey of *effective* user-centered techniques and methods and then show how they could be combined with an informal UML method. The remainder of this chapter presents the survey, describes the current informal UML method, provides a unified approach to scenarios and use cases, introduces the "top ten" user-centered techniques to the UML method, discusses the application of UML to user-centered development, and compares the resulting method with other use case-driven approaches.

9.2 Survey of User-Centered Techniques and Methods

9.2.1 Description of the Survey

A self-selection survey[5] was conducted using three e-mail lists that focus on HCI and usability.

1. The ACM CHI Web list
2. A usability list operated by Clemson University
3. The British HCI Group News Service

Respondents were asked to rate the frequencies with which they employed a number of user-centered techniques, tools, and methods. They were asked to include only those that they found to be effective. These items are summarized in Table 9.1 and described in

[5] I am grateful to Nigel Bevan at Serco Usability Services for his help in drafting the questionnaire for the survey and for helping make a charitable contribution for each response received.

TABLE 9.1 User-Centered Techniques, Tools, and Methods Surveyed*

User-Centered Techniques and Tools	User-Centered Methods
Stakeholder meeting	Design for usability [John Gould]
User analysis/profiling	GUIDE
Personas [Alan Cooper 1999]	LUCID
Task identification	OVID
Comprehensive (e.g., hierarchical) task analysis	STUDIO
Users' conceptual models (of the problem domain)	Usage-centered design/software for use [Constantine and Lockwood 1999]
Contextual analysis	
Evaluate existing system	
Set usability requirements	
Set quantitative usability goals	
Use case analysis	
Essential use cases [Constantine and Lockwood 1999]	
Scenarios of use	
Low-fidelity (e.g., paper) prototyping	
Use of style guides	
Visual interface design	
Navigation design	
Expert (heuristic) usability evaluation	
Informal usability testing	
Formal (e.g., quantitative) usability testing	
Usability checklists	
Attitude questionnaires	
Usability surveys	

*The lists are presented in the order in which they appeared in the questionnaire. For the techniques and tools, this coincides approximately with their order of use within the development process. Methods were presented alphabetically. Respondents were also free to write in other techniques or methods, although very few were mentioned by more than one or two respondents. The results are based on a total of 93 respondents.

detail in Sections 9.2.3 and 9.2.4. Section 9.2.2 presents the ten most popular techniques reported in the survey.

Sixty-five percent of the responses were from the United States; the remainder were international. The majority of respondents were usability practitioners. Slightly fewer

than half of the respondents worked on Web development exclusively, with the bulk of the remainder splitting their efforts between Web and desktop applications. A small number worked exclusively on desktop applications and in other more specialized areas. Further results are shown in Tables 9.2 and 9.3.

TABLE 9.2 Respondents by Country and by Job Function

Respondents by Country, %		Respondents by Job Function, %	
United States	61.3	Usability practitioner	64.5
United Kingdom	12.9	Information architect	7.5
Netherlands	3.2	Developer	6.5
Sweden	3.2	Developer and usability/HCI	6.5
Belgium	2.2	HCI researcher	4.3
France	2.2	Project manager	3.2
Israel	2.2	Technical writer	2.2
Australia	2.2	Software testing/QA	2.2
Canada	2.2	Design director	2.2
Germany	2.2	Technology manager	1.1
Finland	2.2		
Ireland	1.1		
Iraq	1.1		
Italy	1.1		
Mexico	1.1		

TABLE 9.3 Respondents by Product Type and by Time Spent Applying Techniques and Methods

Respondents by Product Type, %		Respondents by Time Spent Applying User-Centered Techniques and Methods, %	
Web only	45.2	All or most	68.8
Mixed Web and desktop	33.3	Less than half	19.4
Desktop only	11.8	Occasionally	11.8
Other	5.4		
Consumer electronics	2.2		
Mobile comm's	1.1		
Multimedia	1.1		

The bulk of the questionnaire contained two lists. The first was a list of common user-centered techniques and tools; the second was a list of user-centered methods. Respondents were asked to rate the frequency of use for each technique, tool, or method that they found to be effective. The results shown in Figure 9.2 have been simplified and show only the percentage of respondents that used each technique or method.

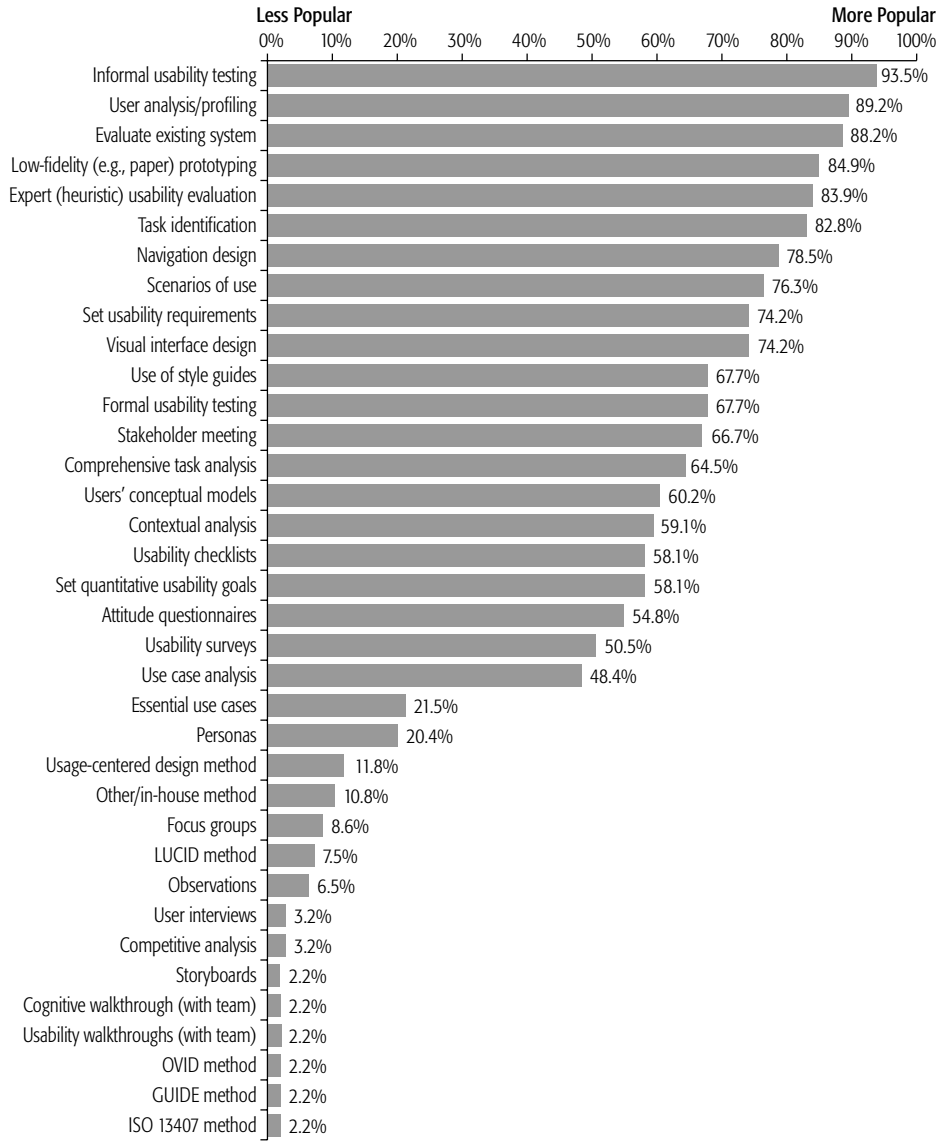


FIGURE 9.2 Percentages of respondents employing user-centered techniques and methods (excludes techniques and methods used by fewer than 2 percent of respondents)

9.2.2 The User-Centered Top Ten

Listed below are the ten most popular user-centered techniques reported in the survey. At least three-quarters of the respondents found them to be effective and reported having used them.

1. Informal usability testing (93.5%)
2. User analysis/profiling (89.2%)
3. Evaluate existing system (88.2%)
4. Low-fidelity (e.g., paper) prototyping (84.9%)
5. Expert (heuristic) usability evaluation (83.9%)
6. Task identification (82.8%)
7. Navigation design (78.5%)
8. Scenarios of use (76.3%)
9. Set usability requirements (74.2%)
10. Visual interface design (74.2%)

Interestingly, with only a few exceptions, these techniques were used in the methods surveyed, as shown in Table 9.4. However, the methods themselves were used infrequently in comparison with the individual techniques.

Listed below are the methods in reverse order of popularity.

1. Usage-centered design (11.8%)
2. Other/in-house method (10.8%)
3. LUCID (7.5%)
4. OVID (2.2%)
5. GUIDE (2.2%)
6. ISO 13407 (2.2%)
7. STUDIO (1.1%)[6]

Because some respondents used more than one method (one reported using six, which I think shows either a lack of perseverance or a surfeit of indecision), the total number using any method was 28 percent. (Bear in mind that respondents defined for themselves what they meant by "used." Many commented that they used only parts of method.) In contrast, 100 percent of the respondents used one or more of the top ten techniques.

So, we have discovered two things. First, Table 9.4 shows that the techniques are not only important individually but also form an important part of user-centered methods (at least those considered here). Second, even though the methods make use of very popular

[6] STUDIO was omitted from Figure 9.2 because it fell below the 2 percent minimum.

TABLE 9.4 Techniques Used in User-Centered Methods

Technique	GUIDE	LUCID	OVID	STUDIO	Usage-Centered Design
Informal usability testing	Yes	Yes	Yes	Yes	Yes
User analysis/profiling	Yes	Yes	Yes	Yes	Yes
Evaluate existing system[a]					
Low-fidelity (e.g., paper) prototyping	Yes	Yes	Yes	Yes[b]	Yes
Expert (heuristic) usability evaluation	Yes	Yes	Yes[c]	No	Yes
Task identification	Yes	Yes	Yes	Yes	Yes
Navigation design	Yes	Yes[d]	Yes	Yes	Yes
Scenarios of use	Yes	Yes	Yes	Yes	No[e]
Set usability requirements	Yes	Yes	Yes	Yes	Yes
Visual interface design	Yes	Yes	Yes	Yes	Yes

[a] None of the methods explicitly mentions the evaluation of existing systems, although they all imply that this would be done as part of user and task analysis.

[b] STUDIO refers to these as superficial designs. They are subject only to heuristic evaluation [Browne 1993, pp. 129–131].

[c] Called a "walkthrough" or an "inspection" in OVID.

[d] LUCID is an approach rather than a method and is not fully specified. However, it does describe the design of the task flow, which is approximately equivalent to navigation design.

[e] Constantine and Lockwood prefer use cases to scenarios in the design phase of development. However, they do use scenarios in usability inspections.

techniques, the methods themselves are not popular by comparison. I am tempted to view this as an indication that more formal methods on the whole are not used, and use this as a motivation for incorporating the techniques into an informal UML method.[7]

9.2.3 User-Centered Techniques

The techniques are described below, in the order in which they appeared in the questionnaire and in Table 9.1.

- *Stakeholder meeting.* Normally held during project inception. A stakeholder is anyone with a vested interest in the project, such as a manager, designer, or user [Rouse 1991].

[7] Although many of the respondents were usability practitioners, they still would have been aware of performing their roles as part of a method if one was being used. Most of the other job functions listed in Table 9.1 would also have had direct involvement with or awareness of a method.

- *User analysis/profiling.* Covers a variety of techniques that involve understanding and describing users [Hackos and Redish 1998, Mayhew 1992].
- *Personas.* Alan Cooper uses this term to represent hypothetical archetypes of real users. Cooper argues that products should be designed for very specific personas, and this approach has been taken up with some interest by the HCI community [Cooper 1999].
- *Task identification.* Also task lists, inventories, and profiles. Identification of the tasks users need to perform, usually as part of task analysis. Task identification needs to be done by interacting with users. Profiling normally includes task frequencies and other characteristics [Hackos and Redish 1998, Mayhew 1999, Shneiderman 1998].
- *Comprehensive (for example, hierarchical) task analysis.* Rigorous analysis of user goals and tasks. Hierarchical task analysis (HTA) is one of the more common forms (see Task identification, above, for references).
- *Users' conceptual models (of the problem domain).* Sometimes referred to simply as a "user model" or, as in Chapter 10, as a "conceptual model." This model is intended to reflect users' understanding of the problem domain, often in the form of conceptual objects and the relationships between them [Norman 1986].
- *Contextual analysis.* Contextual inquiry, ethnography, and related techniques that involve understanding users in their environments [Beyer and Holtzblatt 1998, Mayhew 1999].
- *Evaluate existing system.* A common technique not limited to user-centered design. Sometimes performed as part of contextual analysis [Mayhew 1999].
- *Set usability requirements.* Usually qualitative, as in "The system should provide consistency across components" [Mayhew 1999].
- *Set quantitative usability goals.* Measurable goals, as in "Novice users (defined as first-time users) should take no longer than three minutes to fill in a certain online subscription form" [Mayhew 1999].
- *Use case analysis.* Ivar Jacobson's use cases, as found originally in Objectory and more recently in UML and RUP [Jacobson et al. 1992, 1999].
- *Essential use cases.* A form of abstract use case that focuses on the essence of an interaction between a user and a system [Constantine and Lockwood 1999].
- *Scenarios of use.* Descriptions in almost any form (for example, visual or narrative) of users performing tasks [Carroll 1995].
- *Low-fidelity (for example, paper) prototyping.* Crude mock-ups of user interfaces or portions such as windows, Web pages, or dialogues. Low-fidelity prototypes may be reviewed with users or form the basis of scenarios used to explore interactions [Nielsen 1993].

- *Use of style guides.* Style guides are generally recommended for desktop applications, because they provide consistency within the platform, organization, or product family. They are also becoming popular for Web site design [Mayhew 1999].
- *Visual interface design.* Any deliberate design of the visual aspects of an interface (as opposed to the interface evolving as a side effect of the development process). Visual design is often done in conjunction with low-fidelity prototyping [Nielsen 1995].
- *Navigation design.* This has become better defined with respect to the Web, but it has always been an important part of any user interface. Navigation design determines how users move between Web pages, windows, or dialogues [Hackos and Redish 1998, Fleming 1998].
- *Expert (heuristic) usability evaluation.* Review of a user interface by a usability practitioner familiar with design principles [Nielsen 1993].
- *Informal usability testing.* Usually direct observation, while asking users to think aloud [Tognazzini 1992, Nielsen 1993].
- *Formal (for example, quantitative) usability testing.* More planned or rigorous testing than informal direct observation. May be quantitative, may be set in a usability lab, or may involve detailed analysis of video tapes.
- *Usability checklists.* Primarily employed during evaluation and inspection, but may be used at other times in the development process. They are used to ensure that all desired aspects of usability have been considered [Nielsen and Mack 1994].
- *Attitude questionnaires.* Qualitative (rather than quantitative) questionnaires used to gauge user attitudes toward systems or system components, including individual Web pages or dialogues [Nielsen 1993].
- *Usability surveys.* Conducted to obtain more specific and frequently quantitative data on usability or factors directly affecting usability, such as environment, documentation, and training [Hackos and Redish 1998].

9.2.4 User-Centered Methods

The survey included five existing user-centered software development methods. This was done to determine the popularity of methods (as opposed to techniques).

The methods presented in the questionnaire are described briefly in the following sections. All have been developed from practical experience. Two are British in origin (GUIDE and STUDIO), two are American (LUCID and Usage-Centered Design), and one is Anglo-American (OVID). Most have been published internationally in book form.[8] The methods were presented in alphabetical order.

[8] LUCID is published on the Web and described in Shneiderman [1998]. See Section 9.2.4.2 for references.

9.2.4.1 Graphical User Interface Design and Evaluation (GUIDE)

GUIDE [Redmond-Pyle and Moore 1995] had its origins in the development of large Windows-based applications at LBMS[9] in the late 1980s and early 1990s. It is based largely on practical experience and the participation of one of the authors in the British government's Structured Systems Analysis and Design Method (SSADM) standards working group for graphical user interface (GUI) development.

Figure 9.3 shows the relationship between user-centered GUI design (the shadowed rectangles) and system design. Linkages between system design and GUI design are shown

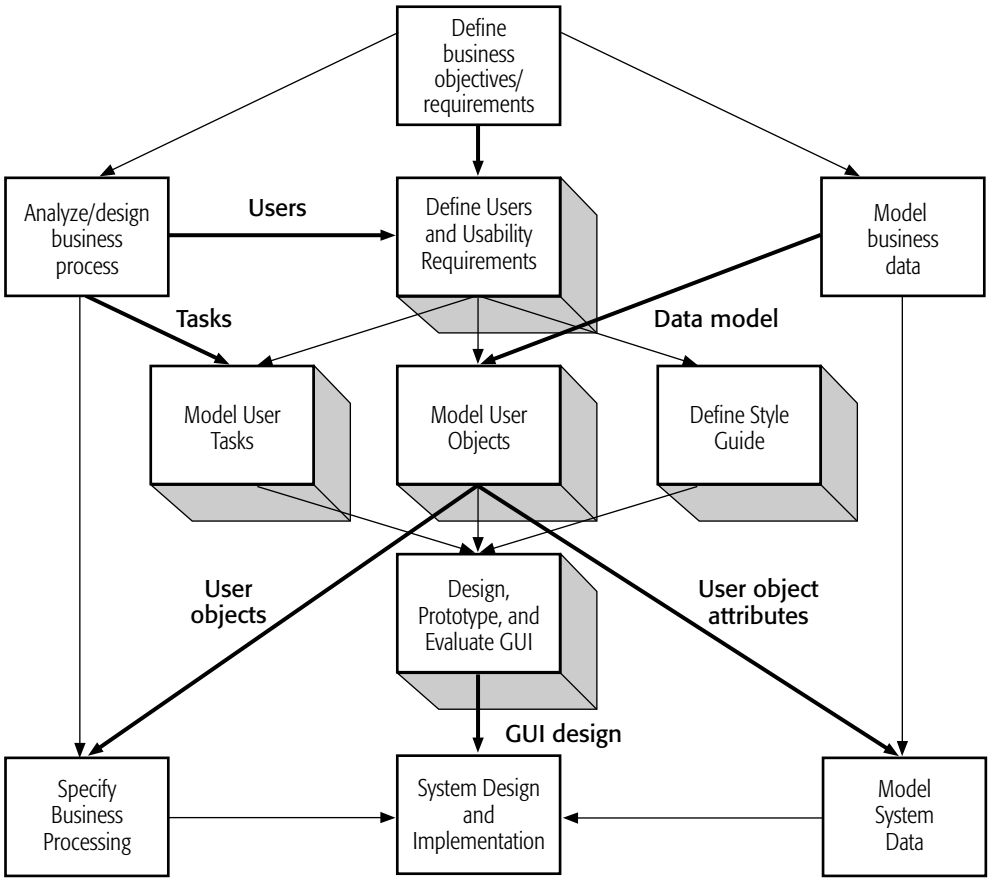


FIGURE 9.3 System and GUI analysis and design in GUIDE (adapted from Redmond-Pyle and Moore [1995, p. 45])

[9] For collectors of acronyms, LBMS stands for Learmonth and Burchett Management Systems.

with heavy arrows and are typical of the kinds of interactions that occur between user-centered and software design activities in many of the methods to be described.

GUIDE includes the following features, which we will come to recognize in this chapter as common to user-centered methods.

- Initial focus on user and task analysis
- Development of a user's conceptual model (partly derived from the business data model in GUIDE)
- Design of the user interface prior to system design

9.2.4.2 Logical User-Centered Interaction Design (LUCID)

The LUCID framework [Kreitzberg 1999] was developed at Cognetics Corporation from its approach to user interface design.[10] LUCID has enjoyed some exposure from the HCI community and appears in books and courses used in the teaching of user interface design (for example [Shneiderman 1998 pp. 104–107]). Its main phases are shown in Table 9.5 (from [Kreitzberg 1999].

LUCID is also used as a basis for the Wisdom method described in see Chapter 6.

9.2.4.3 Object, View, and Interaction Design (OVID)

OVID [Roberts et al. 1998] is one of the earliest user-centered design methods to adopt object-oriented modeling notations. However, rather than using a single domain model, the authors describe three design models based on Don Norman's notion of cognitive engineering [Norman 1986, pp. 45–48]. These models are worth describing in more detail as they are an important concept in user interface design. The relationship between the models is shown in Figure 9.4.

- *Designer's model.* The designer's model[11] expresses the intended user's conceptual model in terms of the objects and relationships that will be represented in the interface. Although the term "designer's model" is in keeping with Norman's original discussion, this is the model that most methods refer to as the user's conceptual model (or user model, or conceptual).

---

[10] This approach was formerly called Quality Usability Engineering (QUE).

[11] The names of the models used here are those found in OVID and in Norman's writing. Chapter 10 discusses more recent terminology used in HCI. There an OVID designer's manual is called a (narrow sense) conceptual model, an OVID programmer's model is called a development model, and an OVID user's conceptual model is called a neutral model.

TABLE 9.5 Overview of LUCID Framework (used with permission of Cognetics Corporation)

Phase	Activity	Description
Concept	Envision	Develop a clear, shared, and communicable vision of the product. Decide on the usability goals for the interface design. Create a "user interface roadmap" to document the preliminary analysis and concepts developed during these activities.
Design	User and task analysis	Perform a comprehensive and systematic analysis of user and task requirements by studying users so as to understand their needs, expectations, tasks, and work processes; determine the implications of this information for the interface.
	Design and prototype	Create a design concept and create a key screen prototype to illustrate it.
	Evaluate and refine	Evaluate the prototype for usability and iteratively refine and expand the design.
Build	Complete detailed design and production	Complete the detailed screen design for the full program. Support late-stage changes.
	Evaluate and refine	Evaluate the complete prototype or early versions of the program for usability and iteratively refine the design.
Release	Release and follow up	Plan and implement the introduction of the product to users, including final usability evaluations to ensure that the product has met the goals established at the beginning of the process. Create and monitor feedback mechanisms to gather data for future releases.

- *Programmer's model.* The programmer's (or implementation) model is most commonly used in object-oriented development methods. It represents the implementation classes used to build the system.
- *User's conceptual model.* This model represents a user's understanding of a system and cannot be directly realized. Because it is not possible to design this model (it is dependent on an individual user's previous experience, for example), the term has come to be used for the *intended* user's conceptual model—that is, the designer's model in OVID. This rather confusing state of affairs is also described in Draper and Norman [1986, pp. 496–497].

OVID's authors make the point, as did Norman, that the user interface must accurately reflect the designer's model in order for users to acquire a suitable conceptual

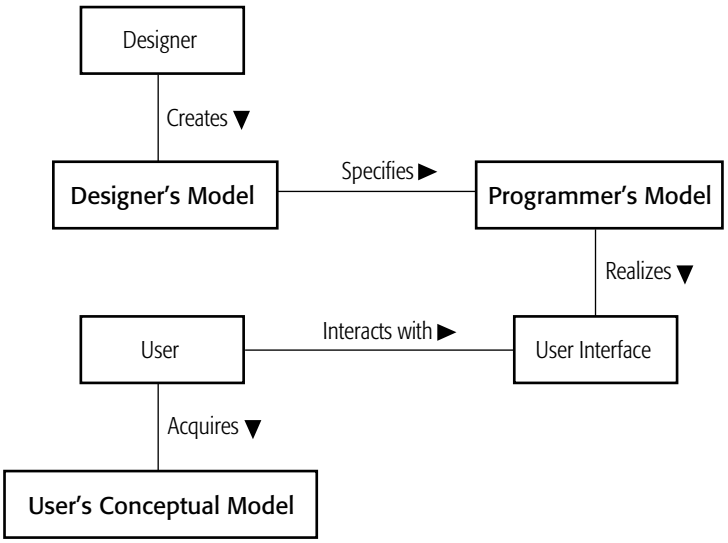


FIGURE 9.4 Relationship between OVID models

model. For this to happen, all aspects of the user interface must be determined by the designer's model, not the programmer's.[12]

The focus of the OVID method is in identifying objects of importance to users, grouping these objects into views that support users' tasks, and detailing the interactions between users and objects. Objects are initially identified through task analysis (although OVID does not prescribe a technique) and organized into a designer's object model, initially based on the user's model. If designed and implemented effectively, the users will understand the designer's model by interacting with the system. The various models are documented using appropriate object-oriented notation. The cycle of activities is shown in Figure 9.5.

Objects are users' conceptual models while *views* are collections of objects that are needed to support users' *tasks*. (OVID does not specify a particular type of task analysis, but use case and hierarchical task analyses are given as examples.) *Interactions* are the actions that are necessary in the interface in order to perform operations on objects.

Like most other user-centered design methods, OVID is iterative and relies on both low- and high-fidelity prototyping.

[12] Norman originally called the user interface the "system image" because it was intended to refer to all aspects of a system that users might experience (including documentation and training). He did not explicitly include the programmer's model.

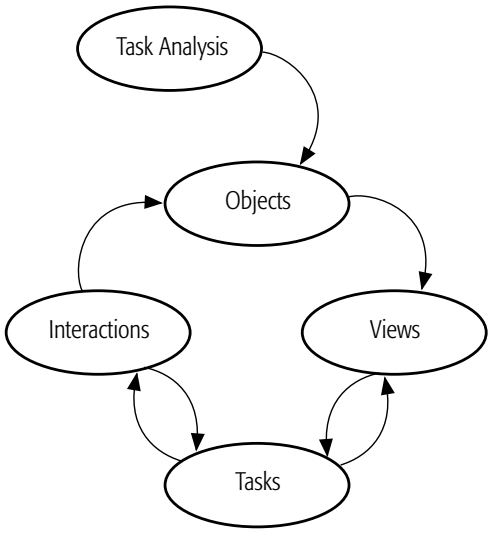


FIGURE 9.5 Cycle of activities in OVID (adapted from Berry et al. [1997]).

9.2.4.4 STructured User-Interface Design for Interaction Optimisation (STUDIO)

STUDIO [Browne 1993] is also based on its author's practical experience in GUI design at KPMG[13] Management Consultants in the early 1990s. Browne advocates using STUDIO for interface-intensive client applications while continuing the use of system-centered development for server-side applications. The main stages of the STUDIO development cycle are as follows.

- *Project planning and proposal.* Cost benefit analysis, quality planning
- *Usability requirements analysis.* Preparing the groundwork, evidence collection, task analysis, validation, reporting of findings
- *Task synthesis.* Task synthesis (convert analysis findings into a user interface design), style guide, design specification, user support documentation, formative evaluation
- *Usability engineering.* Planning, preparation of evaluation materials, prototype build and design audit (in parallel with other activities), prototype evaluation, impact analysis, update specification
- *User interface development.* Hand over specification, integration/interfacing, acceptance testing, termination reporting

[13] Another obscure acronym: Klynveld, Peat, Marwick, and Goerdeler.

STUDIO is very detailed in its description of each activity and its deliverables, but it provides noticeably less guidance on the actual interface design process compared with methods such as GUIDE and usage-centered design. However, in common with GUIDE and LUCID, STUDIO includes the development of a user interface style guide. This approach is not taken with the other methods, but it is useful in ensuring consistent user interface design, especially for large, multi-team projects.

9.2.4.5 Usage-Centered Design

According to Constantine, "usage-centered design focuses on the work that users are trying to accomplish and on what the software will need to supply through the user interface to help them accomplish it" [Constantine 1994, p. 23].[14] Constantine and Lockwood's method includes five key elements:

- Pragmatic design guidelines
- Model-driven design process
- Organized development activities
- Iterative improvement
- Measures of quality

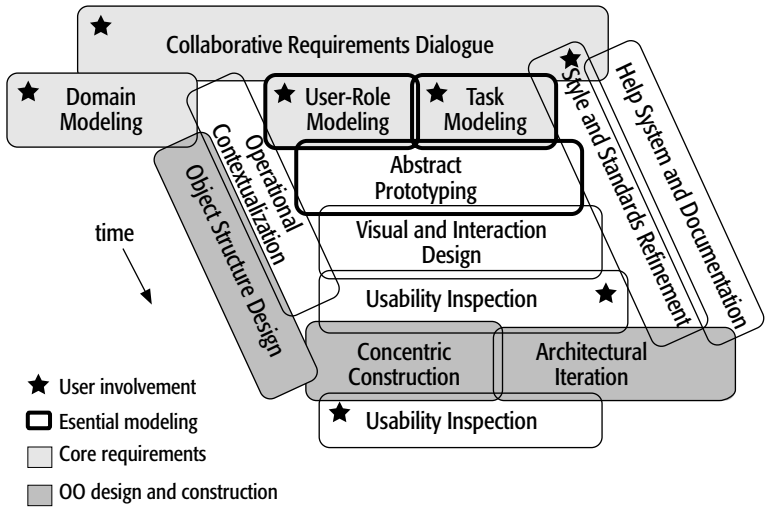


FIGURE 9.6 Usage-centered design activity model (adapted from Constantine and Lockwood [1999], used with permission)

[14] Don't confuse "*user*-centered design" and its acronym (user-centered design) with "*usage*-centered design." The latter is the name of Constantine and Lockwood's method.

These elements are common to many user-centered methods, although the extent to which they are found in individual methods varies. OVID, for example, does not provide pragmatic design guidelines for many of its suggested activities.

As part of the model-driven design process, three core models are used to identify users and their relationships to the system.

1. *Role model.* A collection of user roles and the needs, interests, behaviors, and responsibilities as they apply to each user role.
2. *Task model.* Essential use case model (a form of abstract use case, described in Chapter 7 and Section 9.4.4).
3. *Content model.* An abstract model of users' conceptual objects that is similar to the view model in OVID.

The first two of these models, shown near the top of Figure 9.6, are developed during the initial activities of the process. The content model is established during the abstract prototyping activity and contributes to visual and interaction design.

Other design activities are performed in parallel with the production of the core models. These activities are shown in a diagonal layout in Figure 9.6. Most are self-explanatory, with the exception of operational contextualization. In this activity, Constantine and Lockwood take the approach of adapting the design to the actual operating conditions and environment of the users. (Other methods prefer to resolve this at an earlier stage, although each has its benefits: an early and detailed understanding of context may reduce the design and contextualization effort, while contextualization in parallel with design may provide more realistic feedback.)

Concentric construction (in layers) and architectural iteration form the implementation phase of the process. Usability inspection (which includes a number of usability evaluation techniques) is performed during both design and implementation.

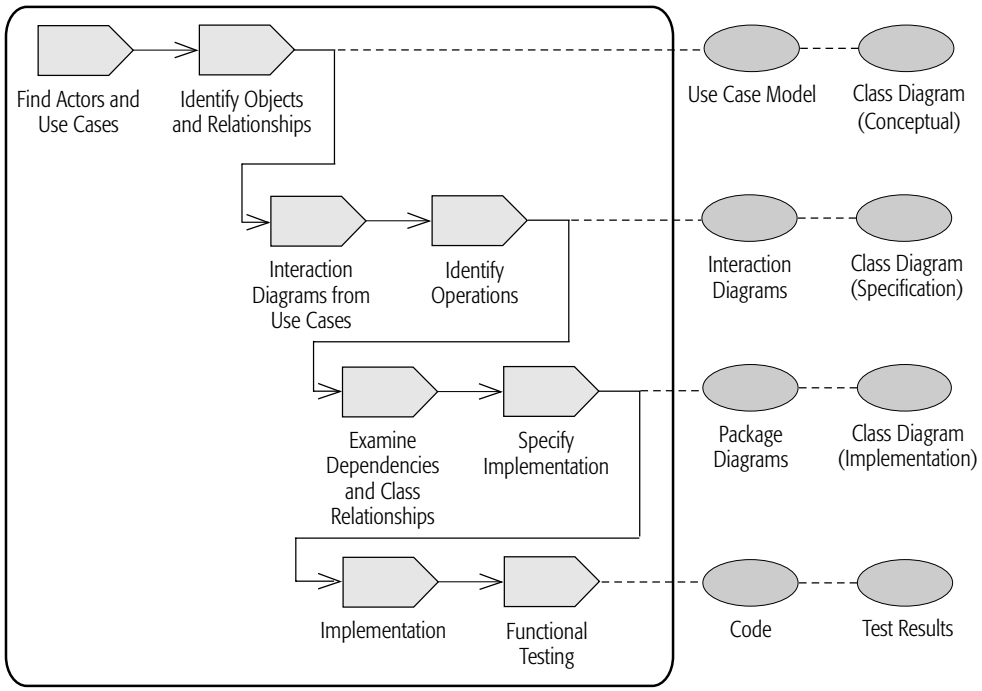
9.3 The Informal UML Method

UML is now published as an industry standard by the Object Management Group [OMG 1999]. It is also the subject of scores of books and papers. Of these, the first edition of Martin Fowler's *UML Distilled* [1997] is familiar to many object-oriented developers, because it was one of the first books to be published, and because it presented a role of UML in a very practical way as an informal method. This is shown in Figure 9.7, using a fairly self-explanatory workflow notation found in the Unified Software Development Process [Jacobson et al. 1999]. The method described here is based on *UML Distilled* with the addition of the implementation and functional testing activities (which UML does not cover and Fowler mentions only briefly).

9.3.1 Perspectives

Fowler introduces the idea of perspectives based on the modeling approach of Cook and Daniels [1994].

- *Conceptual.* The conceptual perspective describes some real or imaginary situation without regard for any software that may be involved.[15] For example, a class diagram with a conceptual perspective is effectively a problem domain model. (If it were based on a user's understanding of the domain, it would be a user's conceptual model, but this issue was not raised by Cook and Daniels or by Fowler.)



Note: Workflow diagrams do not usually show iteration. In common with most other object-oriented methods, the informal UML method *is* iterative, especially in the early activities.

FIGURE 9.7 Informal UML Method

[15] Cook and Daniels used the term "essential" in their original text [Cook and Daniels 1994, p. 12] and said that their use was different from that of some earlier authors. To prevent confusion, I will retain Fowler's substitution of "conceptual."

- *Specification.* Models produced from a specification perspective are more detailed than conceptual models and are somewhat closer to the solution domain.[16] Fowler describes this perspective as looking at the interface between objects[17], not their implementation.
- *Implementation.* Full details of objects (classes), including their detailed behavior and internal representations of state and properties.

Armed with perspectives, Fowler's description of how the UML notation is applied, and my own practical experience in using and teaching UML, we can now consider the informal UML method in a little more detail, as shown in Table 9.6. Simply trying to slot

TABLE 9.6 Description of Informal UML Method by Activity

Activity	Description
Find actors and use cases	Actors and use cases are derived from informal scenarios obtained through discussions with users.
Identify objects and relationships	Objects and relationships are extracted from use cases. Unfortunately, no distinction is made between the domain and user models (i.e., a user's understanding of the domain). The result is a class diagram at the conceptual level, equivalent to a domain model.
Interaction diagrams from use cases	Interaction diagrams elaborate the use cases and aid in the identification of operations on objects (classes).
Identify operations	Each message in an interaction diagram must eventually have a corresponding operation.
Examine dependencies and class relationships	Generalizations (inheritance), aggregations, and associations are identified and refined.
Specify implementation	Iteration and refinement (in increasing detail) finally lead to an implementation-perspective class diagram. Other diagrams may also be used to illustrate object behavior.
Implementation	Coding and unit testing.
Functional testing	Tests that the implemented software performs according to its specification.

[16] It is helpful to view the entire process of software development as bridging a gulf between problem and solution domains.

[17] Classes define objects, but it is convenient to refer to both as objects in many discussions.

user interface design into this process is not very likely to succeed for the reasons that were mentioned in Section 9.1.3.

- Confusion over use cases.
- No separation of user and domain models.
- No deliberate user interface design.
- Lack of contextual information.
- No usability evaluation.

We need to consider these points in more detail.

9.3.2 Confusion over Use Cases

Ivar Jacobson is credited as the inventor of use cases, and his book *Object-Oriented Software Engineering* (OOSE) [Jacobson et al. 1992] is considered the authoritative source on the subject. However, in an earlier paper [Jacobson 1987] he described the importance of use cases in his object-oriented technique, Objectory.

Objectory was intended to allow a system to be built for its users. Jacobson made the point that in order to do this, the behavior of a proposed system must be described. Use cases are the means of achieving this. Figure 9.8 shows a sample use case diagram. The use cases themselves are usually narratives detailing interactions purely from an external perspective.

Even at an early stage, Jacobson was torn between two conflicting applications of use cases, which is a problem that recurs through many discussions up to the present day.

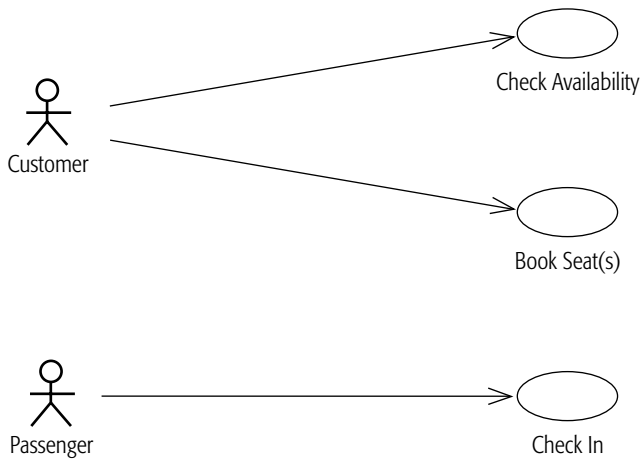


FIGURE 9.8 A use case diagram

First, use cases are a means of describing user interaction with a system from a user's perspective. Second, use cases are a means of describing system behavior from a designer's perspective. Superficially, these two applications may seem to be the same, but this apparent similarity depends on the correspondence of the user's and designer's perspectives.[18] Designers who take the user's perspective into account will write use cases that are much more likely to result in user-centered systems.

In contrast, however, consider this example from Jacobson's paper: "In a telephone exchange, the local calls and long distance calls constitute two different use cases. In both cases the user is an A-subscriber" [Jacobson 1987]. The concepts and terminology being used here are clearly those of a designer, not a user. Users do not consider local and long distance calls as two different tasks. Nor would they identify themselves as "subscribers." If the consequences of this example were to be implemented in a finished telephone system, we could expect a certain degree of user confusion and frustration.

This early description of use cases is very similar to the ones found in UML.

- Use cases constitute a "black box" description of behavior.
- Use cases include *roles* that interact with the systems.
- Roles are adopted by *users*.

Users later became *actors* in object-oriented software engineering (OOSE) and consequently the main focus of attention in use cases (the term "roles" became an abstract concept that did not appear in most use cases). This probably seemed to be an innocent generalization at the time. *Users* are human, but *actors* include any external entity that can adopt a role and interact with the system. In hindsight, this small change created the single most confusing aspect of use cases and certainly detracted from Jacobson's intended purpose of building systems for users. Use cases became, in effect, just another way of describing the dynamic behavior of a system without necessarily providing a clear user focus.

OOSE went on to describe how use cases and objects were different views of the same system [Jacobson et al. 1992, p. 175]. Again, however, this approach failed to acknowledge that a user's view can be very different from a designer's. object-oriented developers are left with the illusion that they are designing systems for users while the reality is that the application of use cases is no guarantee of success.

Following the publication of Jacobson's book, use cases were the subject of much debate and explanation. Jacobson published a series of articles in the *Report on Object-Oriented Analysis and Design* [Jacobson 1994a, 1994b, 1994c, 1994d, 1995b]. In these

[18] Don Norman discusses the relationship between designer and user models, which are closely related to these perspectives [Norman 1986].

articles, he expands on some of the concepts and practicalities of use cases that were not addressed in his book, such as the following.[19]

- *The definition of a good use case.* Jacobson notes that "a good use case when instantiated is a sequence of transactions performed by a system, which yields a measurable result of value for a particular actor" [Jacobson 1994a, p. 17].
- *How and why use cases are created.* Jacobson describes the observation of users in the workplace by interface designers as a key source of use scenarios. He also suggests that use-oriented design and usability testing are very important in understanding the envisioning of the user interface [Jacobson 1994a, p. 17]. Unfortunately, neither of these suggestions has made its way into the common object-oriented practice of generating use cases or into the description of use cases in UML.
- *The relationship between use cases and scenarios.* The term "scenario" is used by the object-oriented community to describe a particular instance of object interaction (given specific states and events): "a specific sequence of actions that illustrates behavior" [Booch et al. 1999, p. 466]. Jacobson observes that a use case class can be modeled as a state machine, with use case instances representing a particular series of states. He describes use scenarios (in the object-oriented sense) and use case instances as equivalent [Jacobson 1994a, p. 17].

Use cases influenced Grady Booch (although he adopted scenarios as an approximately equivalent technique [Booch 1994b, p. 3]). Use cases were incorporated into UML when the Booch, Objectory, and OMT methods were combined [Booch et al. 1999]. However, none of the standard UML references [OMG 1999, Booch et al. 1999, Rumbaugh et al. 1999] describes use cases in enough detail to allow consistent results, let alone provide guidance on user-centered design.

9.3.3 No Separation of User and Domain Models

The second problem I raised with respect to current object-oriented methods is the lack of separation of user and domain models. In HCI, a user's understanding of a system is called the "user's conceptual model" or sometimes just the "user model" or the "conceptual model." Object-oriented approaches tend to ignore the user model and instead concentrate on a domain model, which is intended to represent how the organization or

[19] Much of this material is also included in Jacobson [1995a].

system "really works."[20] Unfortunately, it is extremely rare for users to understand an organization or system in this way. Figure 9.9 shows a user's conceptual model of an elevator. In contrast, Figure 9.10 is a corresponding domain model (although a somewhat simplified one).

Because the user interface for an elevator is so simple, there is little chance that confusing or unnecessary concepts or terms can "leak" from the domain model into the user interface. However, this is a real problem for most software user interfaces.

Another substantial barrier to user-centered development from the domain model is that user interface objects tend to be ignored in the early phases of design. They get "discovered" in later, more detailed stages and a user interface materializes during implementation (see Section 9.3.4).

9.3.4 No Deliberate User Interface Design

The traditional software engineering view of user interface design is that it is an activity that falls outside of its scope. For example, only one of the three undergraduate software engineering textbooks on my shelf has as much as a single chapter devoted to user interface design [Sommerville 1995]. Of the other two, one [Pressman 1997, pp. 409–422] devotes only 24 of 885 pages to the subject, while the third, the most recent of the three [Pfleeger 1998], makes only a few passing references to it. Consequently, in many projects user interfaces evolve as developers discover the need for user interaction. Concepts

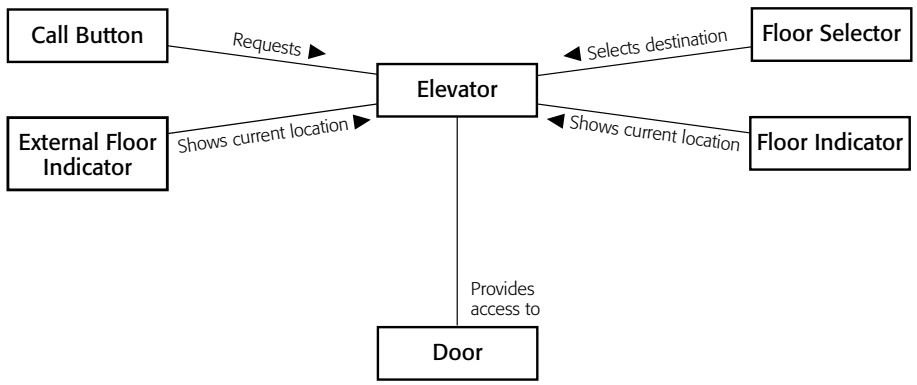
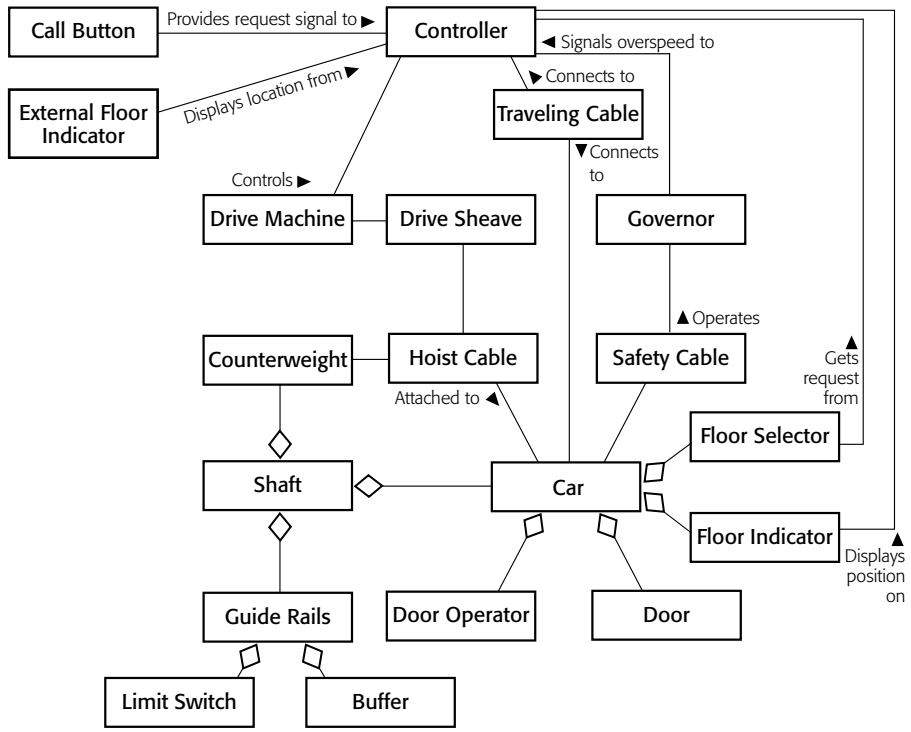


FIGURE 9.9 User's conceptual model of an elevator

[20] Editor's note: the use of "domain model" diverges strongly from the use promoted in Chapter 10 and other chapters, where the domain model is composed of users' referents (including, where required, business referents) and associations between these referents.



Note: Some of the objects (classes) shown in this diagram would be considered by many object-oriented designers to be implementation objects and not part of the domain model. I am not certain that you could get agreement as to which.

FIGURE 9.10 Simplified domain model of an elevator

and terminology in the user interface come straight from the underlying software. I have seen utterly unusable systems developed in this way.

9.3.5 Lack of Contextual Information

Whereas Fowler initially suggested, in his first edition, that use cases be based on interviews with users [Fowler 1997, p. 44], this particular recommendation mysteriously disappeared in the second edition [Fowler 2000]. This is a retrograde step for the development of user-centered systems. User-centered design means understanding the real circumstances or context of a system's use. Sitting in a development office talking to other developers certainly does not provide this. Talking to users is a step in the right direction, but watching and working with users is the best way. (This may not be necessary in all cases, but software development needs to move away from the situation in which developers have no idea what is happening in the real world.)

9.3.6 No Usability Evaluation

Software testing takes many forms and is a science in its own right. Unfortunately, usability evaluation is a completely separate science. Functional testing may prove that a piece of software meets its specifications, but that does not make it suitable for real users in the real world.

Informal usability testing is not difficult to conduct and requires no more than a handful of users [Nielsen 1993, Nielsen and Landauer 1993]. It needs to be an integral part of the design of any interactive system. Heuristic or expert evaluation of designs and prototypes can provide an effective alternative in some cases.

9.4 A Unified Approach to Use Cases and Scenarios

Most of the changes required to make the informal UML method user-centered are relatively straightforward. However, use cases are already part of UML and cannot be considered user-centered in their current form, thus requiring some modification. In contrast, most user-centered design processes are based on scenarios, which are one of the top ten user-centered techniques reported in the survey.

Since Jacobson's introduction of use cases, debates have raged over whether they are more appropriate for design than scenarios. Some authors have tried to avoid the issue by calling them broadly equivalent concepts. However, my own view is that it is preferable to separate the two concepts and to be clear about how they contribute to successful user-centered design. *Both* are required.

In this section we will explore the enigmatic nature of use cases and examine the need for them to be closely integrated with scenarios.

9.4.1 Goal-Based Use Cases

Use cases were not well understood by object-oriented developers in the early 1990s. Part of the confusion arose over Jacobson's lack of formality in defining them (a decision that he later defended as being necessary to their success [Cockburn and Fowler 1998]), but some confusion resulted from the tendency to adapt a good idea to the problem in hand. As a result, the terms "scenario" and "use case" are treated as approximately equivalent by many authors, while various sub-species of use cases have evolved.

In a two-part article, Alistair Cockburn [1997a, 1997b] described some of the variations he encountered in trying to adopt a coherent approach to use cases in object-oriented systems development. In the first part, he shows the variations occurring along the following four dimensions.

1. *Purpose*. Stories of use or requirements. This variation is a result of the confusion surrounding the concepts of use cases and scenarios, as well as the need to describe both existing and proposed systems. Scenarios are better suited to describing stories of use because they describe specific instances. Use cases describe classes of potential scenarios (use case instances) and so are more appropriate for requirements.
2. *Content*. Formal, consistent prose, or contradicting. Formal use cases are typically written in a form of structured English, while consistent prose is informal but self-consistent. Contradicting content is a likely consequence of describing stories of use.
3. *Multiplicity*. One or multiple scenarios per use case. Some use cases are really scenarios, because they describe only a specific use case instance.[21]
4. *Structure*. Unstructured, semiformal, or formal. This dimension may vary with purpose and content or may vary independently.

These dimensions are taken from 18 different types of use cases that Cockburn encountered. He identified <requirements, consistent prose, multiple scenarios, semiformal> as the most useful instance of these dimensions, as well as being the most consistent with Jacobson's original intentions [Jacobson et al. 1992]. Some of the dimensions occurred only in specific combinations. For example, contradicting content must be avoided when describing requirements, but it is an inevitable consequence of documenting stories of actual use.

In the second part of the article, Cockburn introduces three levels at which use cases can operate.

1. *Scope*. System or organization. Use cases can describe the behavior of a single system or of a system of systems (an organization). This is also discussed by Jacobson [1995b]. Organizational use cases are called business use cases in the Unified Software Development Process, which is the successor to Objectory [Jacobson et al. 1999].
2. *Goal specificity*. Summary goals, user goals, or sub-functions. User goals are relevant to system scope use cases. Summary goals can exist at either system or organization scope and are used to organize collections of user goals.
3. *Interaction detail*. Dialogue interface level or semantic interface level. The dialogue interface level describes the syntax interaction (in terms of button

[21] Cockburn uses the term "scenario" in its object-oriented sense.

presses, key clicks, and so on) and should be avoided during requirements gathering. The intent, but not the substance, of the interaction is provided by the semantic interface level.

Most of these dimensions have legitimate applications in system design, but they provide a confusing selection of possibilities for user interface design. Organization scope is useful for identifying actors and assigning roles, but if we assume that the business processes have already been decided, system scope should be our main focus of attention. The goal dimension provides similar choices. Summary goals allow a high-level view of system interaction but are not necessarily useful in designing a user interface. Interface design requires use cases to be presented at the user goal or, in moving further toward detailed design, the sub-function level. Finally, the interaction level determines the amount of detail provided in each use case. Cockburn makes the point that the dialogue level would be inappropriate for requirements analysis because it would commit designers to a specific interface implementation far too early in the process. (He uses "move to the next field" as an example of a dialogue interface description, whereas "enter address" represents a semantic description.)

The complete set of use case dimensions and the values of interest in user interface design are shown in Table 9.7. This combination of attributes gives us use cases that are focused on user goals, excludes unnecessary detail, and is suitable as a statement of requirements (rather than just a story of use). Because their interaction detail is semantic, this particular variety of use case can be described as abstract (as opposed to concrete).

TABLE 9.7 Use Case Dimensions and Values for User Interface Design

Dimension	Value
Purpose	Requirements
Content	Consistent prose
Multiplicity	Multiple scenarios per use case
Structure	Semiformal
Scope	System
Goal specificity	User goals
Interaction detail	Semantic interface level

9.4.2 Scenarios Versus Use Cases

In user-centered design, scenarios usually describe a "rich picture"[22] of interaction—that is, they include not only the information passing through the interface but also a great deal about the context. Scenarios might typically include the following.

- The user's identity, background, experience, and working routine (this also might appear separately in a user profile rather than in the scenario itself)
- Why the user is performing the task at all and why it is being done at this particular time
- The situation or environment in which the interaction is taking place
- Copies of documents used in the interaction and photographs or videos of interactions, user artifacts, and equipment

Some of this might seem mildly absurd for the average office interaction, but it becomes very relevant in factories, warehouses, shop floors, farms, and so on. In fact, as computing becomes more pervasive, scenarios become more important.

Because scenarios are stories of use, what they give us that use cases do not is context—the "who, where, when, and why" of an interaction. As we extract[23] context and generalize the scenarios, we are in effect translating them into use cases, as shown in Figure 9.11.

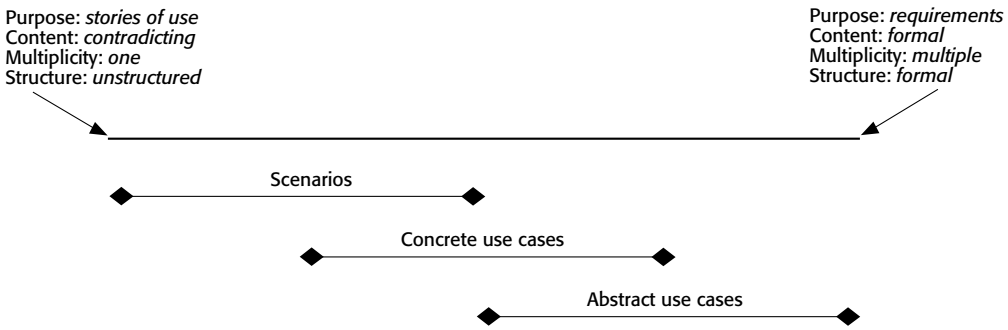


FIGURE 9.11 Relationship between scenarios and use cases

[22] I use the term "rich picture" to mean a broad, multi-faceted view. A rich picture is also a specific technique frequently used in conjunction with scenarios (see [Monk and Howard 1998]). Rich pictures in this latter sense originate from Soft Systems Methodology (SSM) described in [Checkland 1981] and [Checkland and Scholes 1990].

[23] Notice that I say "extract," not "discard." Context is retained as part of the non-functional requirements of design, as we will see in Section 9.4.5.

9.4.3 Context of Use

Entire books have been written about the importance of context in designing interactive systems (see [Beyer and Holtzblatt 1998]). For our purposes, the ISO standard *human-centered design processes for interactive systems* [ISO 1999] will suffice. It identifies the first step in human-centered design as understanding and specifying the context of use.[24] The attributes suggested are shown in Table 9.8 (they are not intended to be exhaustive).

It is useful to split the user context into two parts—user type and role—and then consider the relationships among all four contexts, as shown in Figure 9.12. This figure shows that every *task* is performed in a *role* taken by a *user* within an *environment*. Each of these contexts could have a significant impact on the design of an appropriate user interface. As a result, we are faced with a potentially large number of permutations. Even for a small system, there may be two environments (for example, an office and a customer site), three types of users (an administrative assistant, a sales expert, and manager), and six roles (telephone sales, external sales, and so on). Thus, there can be as many as 36 potential variations per task, although the set of realistic combinations is usually much smaller because not all tasks are performed by all roles taken by all users.

TABLE 9.8 Context of Use from the ISO Standard for Human-Centered Design

Context	Attributes
Tasks	Goals of use of the system, frequency and duration of performance, health and safety considerations, allocation of activities, and operational steps between human and technological resources. Tasks should *not* be described solely in terms of the functions or features provided by a product or system.
Users (for each different type or role)	Knowledge, skill, experience, education, training, physical attributes, habits, preferences, capabilities.
Environments	Social and cultural environment, ambient environment, legislative environment, technical environment, hardware, software, materials, physical and social environments, relevant standards.

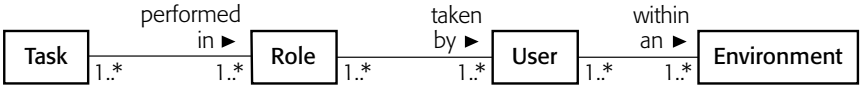


FIGURE 9.12 Relationships among contexts

[24] See Chapter 8 for further discussion of the ISO standard.

9.4.4 Essential Use Cases

Clearly, tasks must be described individually, but a single description is unlikely to be appropriate for all permutations of context. One approach is to factor the user and environment contexts into the role description. This is the solution adopted by Constantine and Lockwood [1999] for their "essential" use cases. It involves providing a separate role for each significant permutation of role, user, and environment and then naming the resulting user role with a descriptive phrase rather than a simple noun. Compare, for example, the role "Customer" with the user roles "Casual Customer," "Web Customer," and "Telephone Customer." Constantine and Lockwood include details of the role itself plus details of its users (referred to as role incumbents) and the environment in each user role description.

The use cases described in UML (little changed from Jacobson's original) are not as straightforward in this respect [Jacobson et al. 1992, Booch et al. 1999]. An actor plays a set of roles, but the roles are not usually described and no mention is made of context or environment. However, this extremely vague state of affairs means that a more user-centered approach, along the lines of essential use cases, is certainly not precluded.

9.4.5 Use Cases as Requirements

It is always tempting to describe requirements analysis as the "what" of software development, with later technical design as the "how." But, as usual, such a simplistic view obscures some important complications. In this case, one complication is that it is almost impossible to say what a system must do without some indication of how it should do it [Davis 1991, Wirfs-Brock 1993]. In other words, we want to describe the proposed *behavior* of a system at a suitable level.[25] In fact, this was Jacobson's original intention for use cases [Jacobson 1987, p. 185].

For interactive systems, we want to arrive at this description of behavior in a way that takes account of its contexts of use. Object-oriented developers are already familiar with the process of starting with a scenario and discarding unnecessary information to arrive at a use case. In the process, however, they discard context. A user-centered approach to this problem is shown in Figure 9.13.

Details of context are extracted from the scenarios and documented separately (as required by the ISO standard [ISO 1999, Section 8.2.2]). The resulting analysis use cases

[25] Even in their most abstract form, use cases describe *behavior* and therefore include some element of implementation detail. The goal in applying use cases to requirements is to exclude inappropriate implementations while allowing designers suitable freedom to innovate.

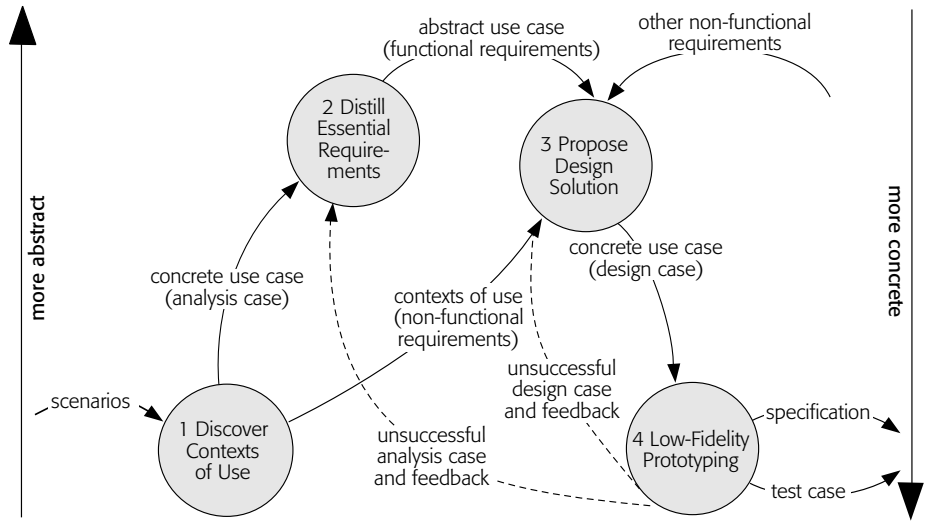


FIGURE 9.13 Relationships among scenarios, use cases, and contexts of use

(or "analysis cases," which is a lot easier to say) are generalized to produce abstract use cases. Being abstract, these use cases describe a whole universe of potential design solutions. Appropriate designs are then synthesized with the help of the contexts of use as well as other non-functional requirements (such as hardware and software platforms or networking constraints). The result is a more concrete version of the abstract use case, which is shown in Figure 9.13 as a design use case or, more simply, "design case." Design cases can be tested by considering specific scenarios, first just in concept, but later as low-fidelity prototypes tested with potential users.

Successful designs are specified in sufficient detail for implementation, with the design cases becoming yet more concrete in the form of test cases.[26] These provide the specific details necessary for functional testing once implementation and unit testing have taken place.

Notice that the approach shown in Figure 9.13 involves starting with concrete information in the form of scenarios, moving to a more general abstract form, and then returning to a concrete design that takes all of the required factors into consideration. This alternation between concrete and abstract is a standard approach to problem solving [Polya 1990] known as the generalize-specialize cycle (see Figure 9.14).

[26] It is not possible to test abstract use cases directly. The test cases take account of a specific design and provide the quality assurance team with the concrete information needed to conduct functional tests.

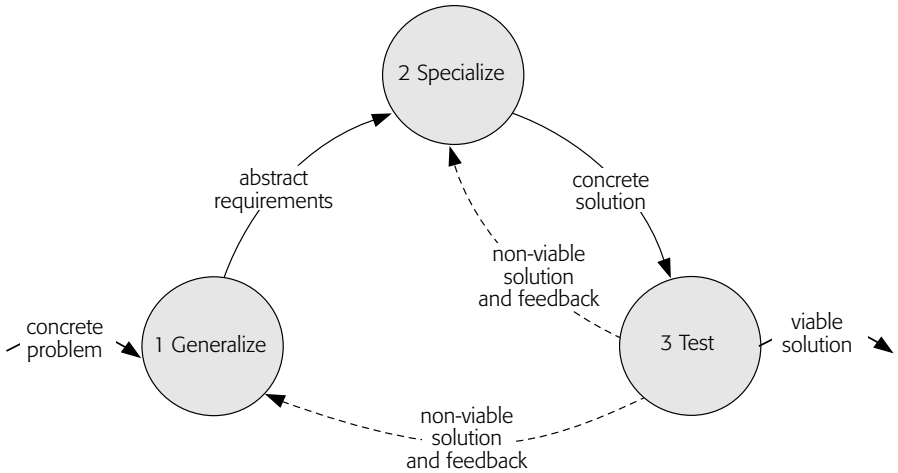


FIGURE 9.14 Generalize-specialize cycle in problem solving

9.5 A User-Centered UML Method

The ISO 13407 human-centered design standard (mentioned in Section 9.4.3) identifies four activities that should take place during system development.

1. Understand and specify the context of use.
2. Specify the user and organizational requirements.
3. Produce design solutions.
4. Evaluate designs against requirements.

So that we are not just arbitrarily adding user-centered techniques to UML, it is useful to make the resultant informal method comply with the ISO 13407 standard. In this section, we will incorporate the "top ten" user-centered methods (from Section 9.2.2) as required to meet the ISO 13407 standard, modify the informal UML method wherever needed (to meet the standard), and describe how the existing UML notation can be applied to user-centered design.

9.5.1 Incorporating the User-Centered Top Ten

Listed below are the ten most popular user-centered techniques from Section 9.2.2. Next to each technique I have indicated whether the technique is necessary in order to satisfy the ISO 13407 standard or to address other shortcomings of the current UML method.

1. *Informal usability testing*. Required in evaluating designs against user requirements.
2. *User analysis/profiling*. Required as part of understanding and specifying the contexts of use.
3. *Evaluate existing system*. Required as part of understanding and specifying the contexts of use.
4. *Low-fidelity prototyping*. Required in order to produce design solutions and to evaluate designs against requirements.
5. *Expert (heuristic) usability evaluation*. Not absolutely essential, but a cost-effective alternative to usability testing in some cases.
6. *Task identification*. Required as part of understanding and specifying the contexts of use.
7. *Navigation design*. Required in order to produce appropriate design solutions and as part of deliberate user interface design.
8. *Scenarios of use*. Required as part of understanding and specifying the contexts of use.
9. *Set usability requirements*. Required in evaluating designs against user requirements.
10. *Visual interface design*. Required in order to produce appropriate design solutions and as part of deliberate user interface design.

In my view, only expert usability evaluation is optional. However, I have still included it in the user-centered UML method because it is a very effective alternative to usability testing (although it cannot entirely replace usability testing). The next step is to group the techniques into activities, as shown in Table 9.9.

The activities are shown in ***bold italics*** as part of a user-centered UML method in Figure 9.15.

9.5.2 Modifying UML for User-Centered Design

Aside from the new activities previously described, the following changes in the informal UML method are needed for user-centered design.

- *Abstract use cases*. The design process needs to be based on abstract use cases as statements of requirements (see Section 9.4). They will be syntax-free, contain no premature design decisions, and be based entirely on a user's view of an interaction. Constantine and Lockwood's essential use cases meet these conditions and also have a structured format [Constantine and Lockwood 1999].

TABLE 9.9 Top Ten User-Centered Techniques Organized as Activities

Techniques	Activity	Description
Scenarios of use Evaluate existing system	Capture/develop scenarios	Observe and interview users in situ, document scenarios, and take photographs, videos, and copies of user artifacts. The scenarios artifact could be just narratives in the simplest case.
User analysis/profiling Task identification Set usability requirements	Contextual analysis	Identify and describe all aspects of context: tasks, users, and environments (see Section 9.4.3).
Navigation design Visual interface design Low-fidelity prototyping	Interface design	Define user's conceptual model; design visual appearance and navigation. Use low-fidelity prototyping to test designs with users. The interface design models artifact will consist of state charts (for navigation), low-fidelity prototypes, and other UML models for full description of user interaction (such as activity diagrams). In addition, as user interface design progresses, new classes will be identified so that the conceptual class diagram (domain model) is produced in parallel.
Expert usability evaluation Information usability testing	Usability evaluation	Evaluate usability before, during, and after implementation. Some usability testing must be done, but expert usability evaluation can be an effective adjunct.

- *User model.* A user's conceptual model is similar to a domain model, but entirely from a user's perspective (see Section 9.3.3). Identification of objects and relationships from the original UML method needs to produce a user model as its initial deliverable, with a conceptual class diagram being produced during user interface design. All concepts and terminology that will appear in the user interface must be consistent with the user model.

These modifications, in combination with the user-centered activities previously proposed, will shift the focus of UML-based software development from system-centered to user-centered.

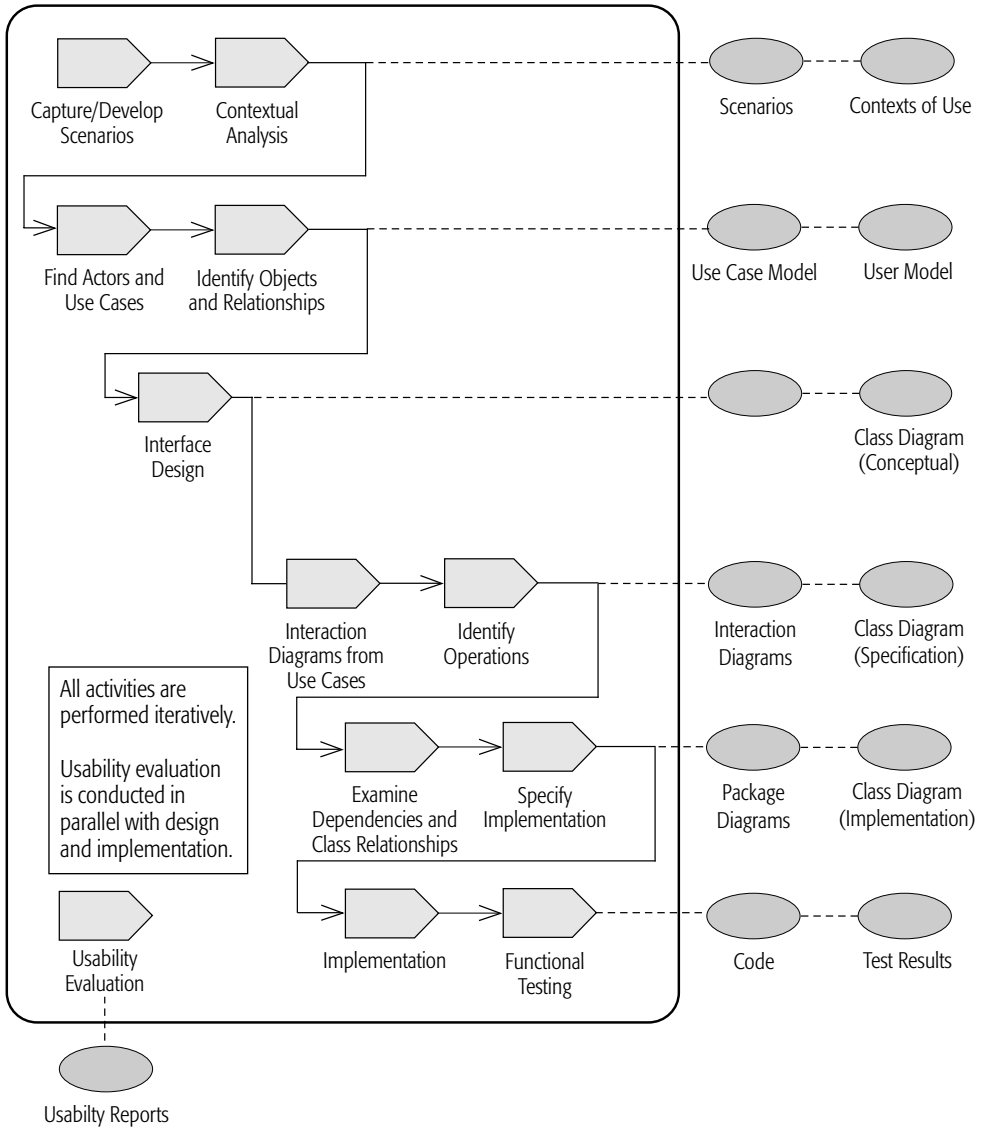


FIGURE 9.15 User-centered UML method

9.5.3 Applying UML Notation to User-Centered Design

The remainder of this section considers how existing UML notation can be used unchanged in user-centered design. One reason for this is that actors may appear where objects (classes) are shown. It is entirely a matter of preference whether actors are shown with the stick-figure representation or as rectangles.

9.5.3.1 Class and Domain Models

Class diagrams document the static relationships between all objects of their defining classes. Their most useful application in the early stages of user interface design is in the production of a user model (user's conceptual model). This is very similar to the domain model described by Fowler [2000], but it is produced entirely from a user's perspective. An example is shown in Figure 9.16.

The user model contains classes that are important from a user's perspective of a system. Some of these classes may not be realized in the detailed design of the system. For example, tickets and boarding cards are important from a user's perspective, but they are simply pieces of paper. They do not necessarily warrant their own classes when the software implementation is considered.

9.5.3.2 Sequence Diagrams

Sequence diagrams have a column showing a "lifeline" for each object involved in the interaction [Fowler 2000]. Early in the design process, the sequence diagrams would reflect the user model and abstract use cases, as shown in Figure 9.17.

9.5.3.3 Collaboration Diagrams

A collaboration diagram contains the same information as a sequence diagram, but it is presented so that the messages making up the interaction annotate the connecting lines (see Figure 9.18). The overall effect makes it more difficult to see the sequence of messages (which are now numbered) but shows the relationships between objects more clearly.

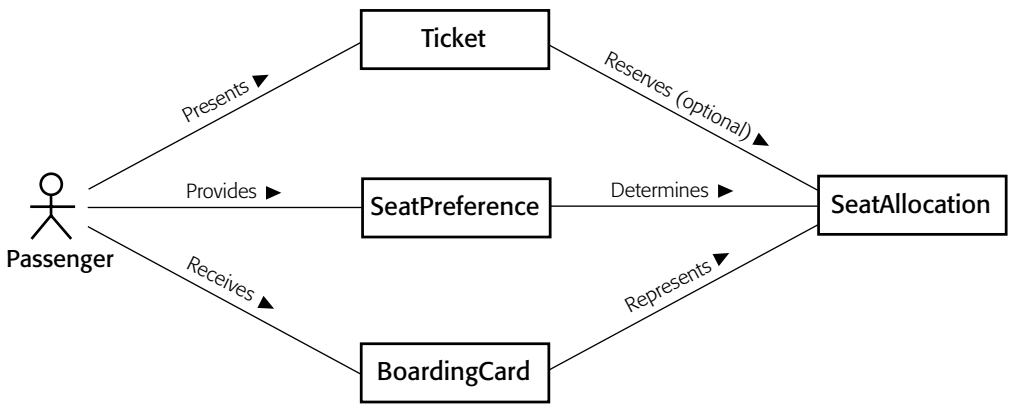


FIGURE 9.16 Simple domain model for passenger check-in (carry-on luggage only)

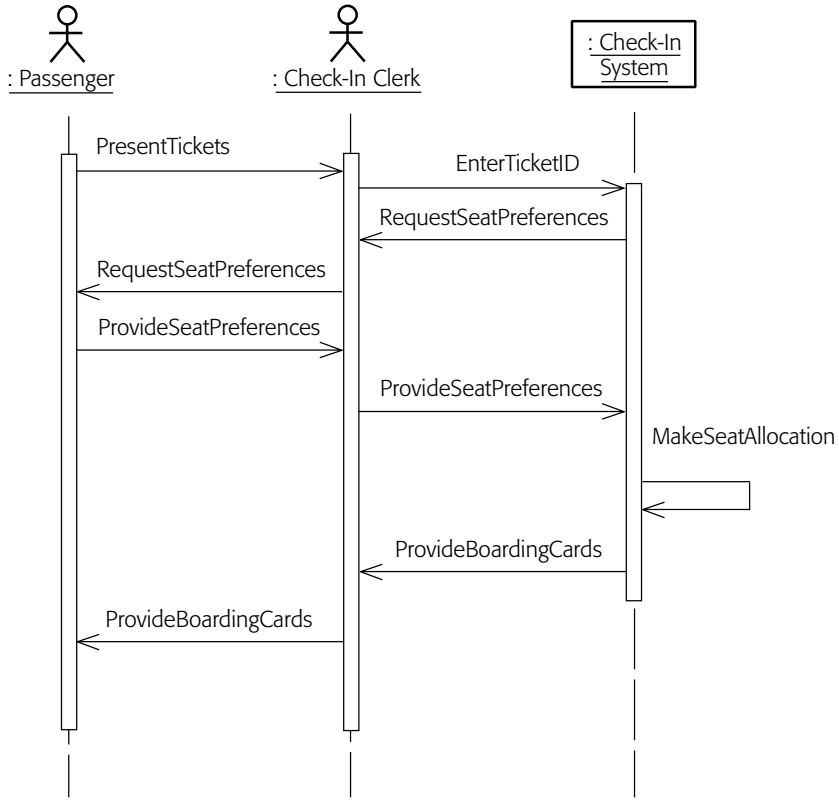


FIGURE 9.17 Sequence diagram for domestic check-in (carry-on luggage only)

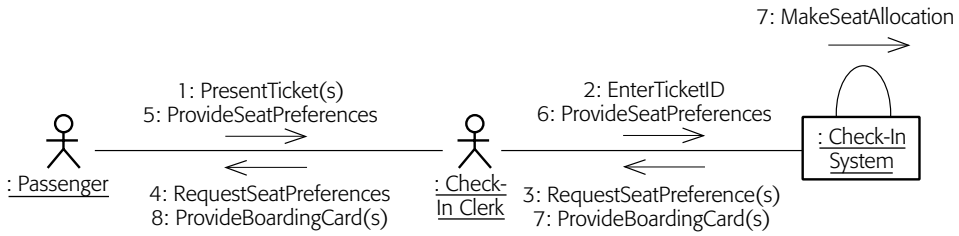


FIGURE 9.18 Collaboration diagram for domestic check-in (carry-on luggage only)

9.5.3.4 State Diagrams

UML's state diagrams are based on state charts used in real-time software engineering [Harel 1987]. Because a user interface can be viewed as a state machine, state diagrams are very appropriate for describing detailed interaction graphically. Perspective is still important, however, so that design decisions will not be taken prematurely. Figure 9.19 shows a simple

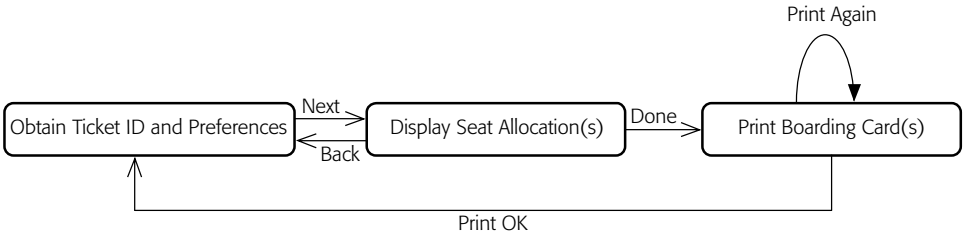


FIGURE 9.19 State diagram for Check-In System class (carry-on luggage only)

state diagram for the check-in example. This diagram has an essential perspective and includes only conceptual objects from the domain model shown in Figure 9.16. A specification perspective would include more detail and would be written with a specific interface design in mind. For example, a specification perspective diagram would show both provisional and final seat allocation activities nested within the appropriate states. While these are not important to the conceptual operation of the user interface, they are very important to its detailed operation. The implementation perspective would add detail relevant to the chosen interface platform (for example, Microsoft Windows, Java AWT, and so on).

9.5.3.5 Activity Diagrams

Activity diagrams combine the features of state charts, flowcharts, and dataflow diagrams (see Yourdon [1989] for descriptions of the latter two types of diagrams). They are activity based, and in their simplest form they resemble a state diagram. However, activity diagrams allow states to be associated with objects by placing states in appropriate columns (called "swimlanes"). Their resemblance to dataflow diagrams comes from the inclusion of an "object flow" notation that connects activities to the objects affected by them. A slightly contrived example is shown in Figure 9.20 (its use is not really warranted in such

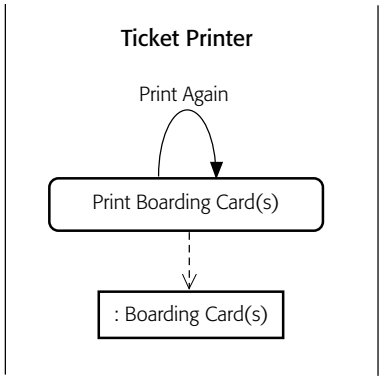


FIGURE 9.20 Simple activity diagram for the Print Boarding Card(s) activity

a simple case). In this example, the Print Boarding Card(s) activity is in the Ticket Printer "lane" and is therefore a state of a Ticket Printer object. The result of the activity is one or more objects of the Boarding Card class.

Because of their structure, activity diagrams are very suitable for modeling workflows.

9.6 Comparisons with Other Use Case-Driven Methods

Attentive readers will recall that use cases were introduced by Ivar Jacobson as part of his Objectory process [Jacobson 1987, Jacobson et al. 1992]. When Jacobson joined Rational Software, Objectory was developed into the Rational Objectory Process and then the Rational Unified Process (RUP).[27]

RUP [Kruchten, 1999] and usage-centered design [Constantine and Lockwood 1999] are software development approaches that describe themselves as use case-driven. Here we briefly compare them with user-centered modifications to the informal UML process, as described in this chapter (UCUML). UCUML is also use case-driven in that abstract use cases describe the desired behavior for the system under design.

9.6.1 Rational Unified Process

RUP is now a proprietary process framework sold by Rational as a Web-based CD-ROM. It is an extremely comprehensive process but does not claim to be user-centered. I have worked briefly with Rational to introduce user-centered design to RUP. Part of this effort was a very educational process for me, as I had not previously worked with such a large process. I discovered that it is *possible* to do user-centered design with RUP, but that developers would need quite a bit of guidance to do so. Some of this guidance appears in the new user-centered design concepts document included as part of the most recent edition of the product, RUP 2000 [Rational 2000].

The following list shows some interesting parallels between RUP and UCUML.

- RUP has a deliberate user interface design activity. This is (described by Kruchten in Chapter 5).
- User interface design in RUP starts with a "use case storyboard" that describes interactions at a high level (see "Guidelines: Use-Case Storyboard" in [Rational 2000]). These interactions are similar to abstract use cases.
- Primitive boundary classes represent users' conceptual objects as they appear in the user interface. While they are not organized into a single model, they

[27] For the history and general approach of this process including details of the Unified Software Development Process (RUP), see [Jacobson et al. 1999].

are the same objects that would appear in the user model suggested for UCUML.

- Contexts of use are established, but across a number of activities and artifacts, as shown in Table 9.10.

The differences between RUP and UCUML are as follows.

- RUP does not distinguish between domain and user models.
- Use cases in RUP are vaguely defined (except for the advice given as part of user interface design). Scenarios are viewed only as instances of use cases.

TABLE 9.10 Contexts of Use in RUP 2000 (from [Rational 2000], used with permission)

Context	RUP Artifact
Environments	High-level: ■ Business Vision [Section: Customer Environment] ■ Stakeholder Requests ■ Vision [Section: User Environment]
Users	High-level: ■ Business Vision [Section: Customer Profiles] ■ Stakeholder Requests ■ Vision [Section: User Profiles]
Roles	High-level: ■ Business Actor (external users) ■ Business Worker (internal users) Detailed: ■ Actor
Tasks	High-level: ■ Stakeholder Requests ■ Vision [Section: Product Features] Detailed: ■ Use-Case Storyboard ■ Use Case

- Usability evaluation is presented only as part of user interface prototyping. It is not considered to be part of overall system testing.
- RUP is an elaborate and very detailed process for software construction from beginning to end. By contrast, UCUML is a lightweight approach that gives only the general flavor of how *user-centered* software development should be done.

In reality, it is unlikely that an organization would be considering the extremes of a full-blown process such as RUP and a lightweight method such as UCUML. However, usage-centered design might provide a viable compromise for those in search of detailed guidance.

9.6.2 Usage-Centered Design

Usage-centered design [Constantine and Lockwood 1999] is described briefly in Section 9.2.4.5. Unlike RUP, usage-centered design *does* have a user-centered philosophy. Consequently, it has a number of points in common with the other user-centered processes considered in this chapter.

Features common to both usage-centered design and UCUML are as follows.

- *Abstract use cases.* Constantine and Lockwood refer to their particular form of abstract use cases as essential use cases (described in Section 9.4.4).
- *Deliberate user interface design.* Usage-centered design includes an interface content modeling activity that concerns itself with both navigation and content. The implementation modeling activity deals with visual design.
- *Usability evaluation.* Expert evaluations and usability testing are both included in usage-centered design.

The main differences in approach center around scenarios and contexts of use.

- *Scenarios.* Constantine and Lockwood do not value scenarios during analysis and design. They appear to be concerned that scenarios contain too much extraneous information [Constantine and Lockwood 1999, p. 106].
- *Contexts of use.* Usage-centered design does not directly address the issues of contexts of use as presented in the ISO 13407 standard (see Section 9.4.3). Confusingly, Constantine and Lockwood use the term "context" both in the sense that I have used it in this chapter and as the state that the user interface is in at a particular moment of interaction. Contextual inquiry [Beyer and Holtzblatt 1998] is mentioned but is not directly drawn into the method.

However, to be fair to usage-centered design, it is a complete method whereas UCUML is a skeletal approach. In addition, Constantine and Lockwood's description of their method [Constantine and Lockwood 1999] includes a large amount of information and advice on user interface and user-centered design. This provides developers with most of what is necessary to design user-centered systems, if only they can be persuaded to leave the mainstream approaches behind.

9.7 Conclusions

9.7.1 The Benefits

At the beginning of this chapter, I suggested that the benefits resulting from a convergence of UML and user-centered design would be similar to those derived from the widespread adoption of UML itself. Each of the following benefits applies to this convergence.

- *Increased awareness of user-centered issues among developers.* Developers do not set out to create unusable systems. It is naïveté rather than malice that leads to the usability problems with which we are all too familiar. Making user-centered design an integral part of the software development process is probably the most practical way of overcoming the current lack of awareness.
- *Better communication between designers and developers.* This benefit applies as much to user interface designers as it does to system designers. Conflicts frequently arise during development when there are differing goals. A common vision of how scenarios and use cases are captured and developed, a common notation in expressing a variety of design models, and a common understanding of usability evaluation would prevent some of these conflicts.
- *Skills are more readily transferred between projects.* If user-centered design became part of "mainstream" software development for interactive systems, there would be less difficulty in hiring and educating staff. Developers leaving one environment would not need to be retrained for another employer or project using alternative techniques or methods.
- *Improved development support from software tools.* User-centered methods currently have little support through software tools in comparison with UML. Adopting common notations (where possible) and incorporating user-centered techniques into UML would significantly improve this situation.

9.7.2 The Challenges

Still, combining user-centered design with UML raises a number of issues.

- *Lack of skills.* Although the transfer of skills between projects will eventually be an advantage to a unified approach, initially there will be a shortage. This will be particularly true for small projects in which developers will be expected to perform various roles, including those of a usability practitioner.
- *Lack of specialization.* Some members of the HCI community argue that usability and user interface design activities must be performed by specialists. While this is certainly true in complex cases, I have seen projects that would have benefited from *any* application of user-centered principles. In addition, greater awareness of user-centered design is likely to lead to greater involvement of usability practitioners in situations that require it, either in the direct performance of user-centered design activities or as facilitators, mentors, and educators.
- *False security.* Part of the argument over specialization is whether organizations would be deluding themselves that they were being user-centered when they were simply applying a few user-centered techniques badly. However, with disparate goals among members of the development team, pressing deadlines, and unenlightened management, we have pretty much the same result on many projects already. User interface designers fail to persuade developers to implement user-centered designs, contextual analysis is abandoned as having a detrimental effect on schedules, and usability tests are ignored for myriad reasons. Getting the entire development team pulling together in the same direction would reduce the acceptability of user-centered shortsightedness.
- *Lack of aptitude.* To make a long story short, the traditional wisdom is that user-centered design requires people skills, and software development requires engineering skills. We see developers who find it hard to accept that user-centered design is important, and we assume that this occurs because they do not have an aptitude for user-centered design. However, since most developers have not been taught user-centered design as an integral part of software engineering courses, it is hardly surprising that they view user-centered design activities as eccentric and not really relevant to them. Some developers may become very proficient at user-centered design, whereas others may not.
- *Lack of separation.* There is a view that in an ideal world, user-centered design would take place without the possibility of "contamination" from the software development process. Separate team members, tools, and notations would reduce the possibility of implementation concepts "leaking" into the user interface. At best, this argument is unrealistic in comparison with everyday practice.

In most cases, user-centered design is not being done at all, and the user interface concepts *are* the implementation concepts with a few alterations where sufficient complaints have been received from users.

- *Conflicts of interest.* Very few of us have the luxury of being single-minded in purpose. Deadlines, costs, resources, scheduling, and the status quo all conspire to make design decisions much more complex than we would like them to be. As both a developer and a user-centered designer, I frequently find myself having to weigh the usability of the design against the difficulty of its implementation. Is this a bad thing? I think not, for the following reasons:

  - It is a well-informed deliberation. I cannot hide from or overlook important issues in the hope of swaying the decision one way or another.
  - I own the problem. I cannot dismiss one side of the argument or the other just because it falls outside my area of concern.
  - I understand the impact of poor usability. Because I work with users, I empathize with their problems. Since I understand that software does not have to be inherently difficult to use, I have a low tolerance for design decisions that result in poor usability.

Am I uniquely gifted in being able to work both as a usability practitioner and as a developer? I seriously doubt it. All that it requires is an acceptance that users and user-centered design are important to the development of usable software.

9.7.3 The Future

This chapter has not been about introducing a new method for user-centered design that can be placed on the bookshelf and forgotten. I have proposed the convergence of user-centered design with UML—perhaps the most popular development technology ever known to the software industry. Just as UML has become the de facto standard for object-oriented development, UCUML needs to become the de facto standard for interactive systems. This might be achieved in part by the development of an extension of UML in the same way that real-time variations [Douglas 1999] and Web variations [Conallen 2000] of UML have arisen.

User-centered design needs to be adopted and taught as an integral part of interactive software development. It must not continue to be a specialized activity unknown to the majority of the software industry.

Finally, I encourage anyone working in a UML-based interactive development environment to try to incorporate some of the ideas presented here. This needs to be done not only by introducing user-centered techniques but also by trying to make the entire development philosophy user-centered. Please let me know of your successes or failures.

9.8 References

[Berry et al. 1997] D. Berry, S. Isensee, and D. Roberts. Designing for the User with OVID: Bridging User Interface Design and Software Engineering. Found at the IBM Corporation Web site, at *http://www.ibm.com/ibm/hci/guidelines/design/ovida.html*.

[Beyer and Holtzblatt 1998] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. San Francisco: Morgan Kaufmann, 1998.

[Booch 1994a] G. Booch. *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1994.

[Booch 1994b] G. Booch. The Booch Method: Scenarios. *Report on Object-Oriented Analysis and Design,* 1 (3), 1994, 3–6.

[Booch et al. 1999] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

[Browne 1993] D. Browne. *STUDIO: STructured User-Interface Design for Interaction Optimisation*. New York: Prentice Hall, 1993.

[Carroll 1995] J. M. Carroll, ed. *Scenario-Based Design: Envisioning Work and Technology in System Development*. New York: John Wiley and Sons, 1995.

[Checkland 1981] P. Checkland. *Systems Thinking, Systems Practice*. Chichester: John Wiley and Sons, 1981.

[Checkland and Scholes 1990] P. Checkland and J. Scholes, *Soft Systems Methodology in Practice*. Chichester: John Wiley and Sons, 1990.

[Cockburn 1997a] A. Cockburn. Goals and Use Cases. *JOOP,* 10 (6), 1997, 35–40.

[Cockburn 1997b] A. Cockburn. Using Goal-Based Use Cases. *JOOP,* 10 (7), 1997, 56–62.

[Cockburn and Fowler 1998] A. Cockburn and M. Fowler. Question Time! About Use Cases, *Proceedings of the OOPSLA Conference*. New York: ACM, 1998, 226–243.

[Conallen 2000] J. Conallen. *Building Web Applications with UML*. Reading, MA: Addison-Wesley, 2000.

[Constantine 1994] L. L. Constantine. Essentially Speaking. *Software Development,* 2 (11), 1994, 95–96.

[Constantine 1995] L. L. Constantine. Essential Modeling. *Interactions,* 2 (2), 1995, 34–46.

[Constantine and Lockwood 1999] L. L. Constantine and L. A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. New York: ACM Press, 1999.

[Cook and Daniels 1995] S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* New York: Prentice Hall, 1994.

[Cooper 1999] A. Cooper. *The Inmates Are Running the Asylum*. Indianapolis: SAMS Publishing, 1999.

[Davis 1991] A. M. Davis. *Software Requirements Analysis and Specification*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[Douglas 1999] B. P. Douglas. *Real-Time UML: Developing Efficient Objects for Embedded Systems,* 2nd Ed. Reading, MA: Addison-Wesley, 1999.

[Draper and Norman 1986] S. W. Draper and D. Norman, eds. *User Centered System Design*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1986.

[Fleming 1998] J. Fleming. *Web Navigation: Designing the User Experience*. Sebastopol, CA: O'Reilly and Associates, 1998.

[Fowler 1997] M. Fowler. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1997.

[Fowler 2000] M. Fowler. *UML Distilled: Applying the Standard Object Modeling Language*. 2nd Ed., Reading, MA: Addison-Wesley, 2000.

[Graham 1997] I. Graham. Some Problems with Use Cases . . . and How to Avoid Them. In D. Patel, Y. Sun, and S. Patel, eds. *Oois '96: 1996 International Conference on Object Oriented Information Systems*. London: Springer, 1997, 18–27.

[Graham et al. 1997] I. Graham, B. Henderson-Sellers, and H. Younessi. *The OPEN Process Specification*. Harlow, England: Addison-Wesley, 1997.

[Hackos and Redish 1998] J. T. Hackos and J. C. Redish. *User and Task Analysis for Interface Design*. New York: Wiley and Sons, 1998.

[Harel 1987] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming,* 8 (3), 1987, 231–274.

[ISO 1999] International Standardization Organization. *ISO 13407. Human-Centred Design Processes for Interactive Systems*. Geneva, Switzerland: ISO, 1999.

[Jacobson 1987] I. Jacobson. Object Oriented Development in an Industrial Environment. In *Proceedings of the OOPSLA '87 Conference,* New York: ACM, 1987, 183–191.

[Jacobson 1994a] I. Jacobson. Basic Use-Case Modeling. *Report on Object-Oriented Analysis and Design,* 1 (2), 1994, 15–19.

[Jacobson 1994b] I. Jacobson. Basic Use-Case Modeling (Continued). *Report on Object-Oriented Analysis and Design,* 1 (3), 1994, 7–9.

[Jacobson 1994c] I. Jacobson. Toward Mature Object Technology. *Report on Object-Oriented Analysis and Design,* 1 (1), 1994, 36–39.

[Jacobson 1994d] I. Jacobson. Use Cases and Objects. *Report on Object-Oriented Analysis and Design,* 1 (4), 1994, 8–10.

[Jacobson 1995a] I. Jacobson. The Use-Case Construct in Object-Oriented Software Engineering. In J. M. Carroll, ed. *Scenario-Based Design*. New York: John Wiley and Sons, 1995, 309–336.

[Jacobson 1995b] I. Jacobson. Use Cases in Large-Scale Systems. *Report on Object-Oriented Analysis and Design,* 1 (6), 1995, 9–12.

[Jacobson et al. 1992] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: ACM Press, 1992.

[Jacobson et al. 1999] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

[Kreitzberg 1999] C. Kreitzberg. The LUCID Design Framework. Found at the Cognetics Corporation Web site, at *http://www.cognetics.com/lucid/lucid2aoverview.pdf*.

[Kruchten 1999] P. Kruchten. *The Rational Unified Process—An Introduction*. Reading, MA: Addison-Wesley, 1999.

[Mayhew 1992] D. J. Mayhew. *Principles and Guidelines in Software User Interface Design.* Englewood Cliffs, NJ: Prentice Hall, 1992.

[Mayhew 1999] D. J. Mayhew. *The Usability Engineering Lifecycle: A Practitioner's Handbook for User Interface Design.* San Francisco: Morgan Kaufmann, 1999.

[Monk and Howard 1998] A. Monk and S. Howard. Methods & Tools: The Rich Picture. *Interactions,* March and April, 1998, 21–30.

[Nielsen 1993] J. Nielsen. *Usability Engineering.* San Diego: Academic Press, 1993.

[Nielsen 1995] J. Nielsen. Scenarios in Discount Usability Engineering. In J. M. Carroll, ed. *Scenario-Based Design.* New York: John Wiley and Sons, 1995, 59–84.

[Nielsen and Landauer 1993] J. Nielsen and T. K. Landauer. A Mathematical Model of the Finding of Usability Problems. *Proceedings of ACM INTERCHI '93,* (Amsterdam, The Netherlands), New York: ACM, 1993, 206–213.

[Nielsen and Mack 1994] J. Nielsen and R. L. Mack. *Usability Inspection Methods.* New York: John Wiley and Sons, 1994.

[Norman 1986] D. Norman. Cognitive Engineering. In D. Norman and S. W. Draper, eds. *User Centered System Design.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1986, 31–71.

[Olson and Moran 1995] J. Olson and T. Moran. Mapping the Method Muddle: Guidance in Using Methods for User Interface Design. In M. Rudisill, C. Lewis, P. Polson, and T. McKay, eds. *Human-Computer Interface Design: Success Cases, Emerging Methods and Real-World Context.* San Francisco: Morgan Kaufmann, 1995.

[OMG 1999] Object Management Group. *OMG Unified Modeling Language Specification.* Version 1.3, Needham, MA: OMG, 1999. Found at *http://www.omg.org.*

[Pfleeger 1998] S. L. Pfleeger. *Software Engineering: Theory and Practice.* Upper Saddle River, NJ: Prentice Hall, 1998.

[Polya 1990] G. Polya. *How to Solve It: A New Aspect of Mathematical Method.* 2nd Ed., London: Penguin Books, 1990.

[Pressman 1997] R. Pressman. *Software Engineering: A Practitioner's Approach.* 4th Ed., New York: McGraw-Hill, 1997.

[Rational 2000] Rational Software Corporation. *Rational Unified Process 2000.* Cupertino, CA: Rational Software Corporation, 2000.

[Redmond-Pyle and Moore 1995] D. Redmond-Pyle and A. Moore. *Graphical User Interface Design and Evaluation.* London: Prentice Hall, 1995.

[Roberts et al. 1998] D. Roberts, D. Berry, S. Isensee, and J. Mullaly. *Designing for the User with OVID: Bridging User Interface Design and Software Engineering.* Indianapolis: Macmillan Technical Publishing, 1998.

[Rouse 1991] W. B. Rouse. *Design for Success.* New York: John Wiley and Sons, 1991.

[Rumbaugh et al. 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall, 1991.

[Rumbaugh et al. 1999] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Reading, MA: Addison-Wesley, 1999.

[Schneiderman 1998] B. Schneiderman. *Designing the User Interface,* 3rd Ed. Reading, MA: Addison-Wesley, 1998.

[Sommerville 1995] I. Sommerville. *Software Engineering,* 5th Ed. Harlow, England: Addison-Wesley, 1995.

[Tognazzini 1992] B. Tognazzini. *Tog on Interface*. Reading, MA: Addison-Wesley, 1992.

[Wieringa 1998] R. Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys,* 30 (4), 1998, 459–528.

[Wirfs-Brock 1993] R. Wirfs-Brock. Designing Scenarios: Making the Case for a Use Case Framework. *Smalltalk Report,* November–December, 1993, 9–20.

[Yourdon 1989] E. Yourdon. *Modern Structure Analysis*. Englewood Cliffs, NJ: Prentice Hall International, 1989.