4

Capturing the Requirements

In this chapter, we look at

- Eliciting requirements from our customers
- Modeling requirements
- Reviewing requirements to ensure their quality
- Documenting requirements for use by the design and test teams

In earlier chapters, when looking at various process models we noted several key steps for successful software development. In particular, each proposed model of the software development process includes activities aimed at capturing requirements: understanding our customers' fundamental problems and goals. Thus, our understanding of system intent and function starts with an examination of requirements. In this chapter, we look at the various types of requirements and their different sources, and we discuss how to resolve conflicting requirements. We detail a variety of modeling notations and requirements specification methods, with examples of both automated and manual techniques. These models help us understand the requirements and document the relationships among them. Once the requirements are well understood, we learn how to review them for correctness and completeness. At the end of the chapter, we learn how to choose a requirements-specification method that is appropriate to the project under consideration, based on the project's size, scope, and the criticality of its mission.

Analyzing requirements involves much more than merely writing down what the customer wants. As we shall see, we must find requirements on which both we and the customer can agree and on which we can build our test procedures. First, let us examine exactly what requirements are, why they are so important (see Sidebar 4.1), and how we work with users and customers to define and document them.

4.1 THE REQUIREMENTS PROCESS

When a customer requests that we build a new system, the customer has some notion of what the system should do. Often, the customer wants to automate a manual task, such as paying bills electronically rather than with hand-written checks. Sometimes, the customer wants to enhance or extend a current manual or automated system. For example, a telephone billing system that had charged customers for only local telephone service and long-distance calls may be updated to bill for call forwarding, call waiting, and other new features. More and more frequently, a customer wants products that do things that have never been done before: tailoring electronic news to a user's interests, changing the shape of an airplane wing in mid-flight, or monitoring a diabetic's blood sugar and automatically controlling insulin dosage. No matter whether its functionality is old or new, a proposed software system has a purpose, usually expressed in terms of goals or desired behavior.

SIDEBAR 4.1 WHY ARE REQUIREMENTS IMPORTANT?

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

(Brooks 1987)

In 1994, the Standish Group surveyed over 350 companies about their over 8000 software projects, to find out how well they were faring. The results are sobering. Thirty-one percent of the software projects were canceled before they were completed. Moreover, in large companies, only 9% of the projects were delivered on time and within budget; 16% met those criteria in small companies (Standish 1994). Similar results have been reported since then; the bottom line is that developers have trouble delivering the right system on time and within budget.

To understand why, Standish (1995) asked the survey respondents to explain the causes of the failed projects. The top factors were reported to be

1. Incomplete requirements (13.1%)
2. Lack of user involvement (12.4%)
3. Lack of resources (10.6%)
4. Unrealistic expectations (9.9%)
5. Lack of executive support (9.3%)
6. Changing requirements and specifications (8.7%)
7. Lack of planning (8.1%)
8. System no longer needed (7.5%)

Notice that some part of the requirements elicitation, definition, and management process is involved in almost all of these causes. Lack of care in understanding, documenting, and managing requirements can lead to a myriad of problems: building a system that solves the wrong problem, that doesn't function as expected, or that is difficult for the users to understand and use.

Furthermore, requirements errors can be expensive if they are not detected and fixed early in the development process. Boehm and Papaccio (1988) report that if it costs $1 to find and fix a requirements-based problem during the requirements definition process, it can cost $5 to repair it during design, $10 during coding, $20 during unit testing, and as much as $200 after delivery of the system! So it pays to take time to understand the problem and its context, and to get the requirements right the first time.

A **requirement** is an expression of desired behavior. A requirement deals with objects or entities, the states they can be in, and the functions that are performed to change states or object characteristics. For example, suppose we are building a system to generate paychecks for our customer's company. One requirement may be that the checks be issued every two weeks. Another may be that direct deposit of an employee's check be allowed for each employee at a certain salary level or higher. The customer may request access to the paycheck system from several different company locations. All of these requirements are specific descriptions of functions or characteristics that address the general purpose of the system: to generate paychecks. Thus, we look for requirements that identify key entities ("an employee is a person who is paid by the company"), limit entities ("an employee may be paid for no more than 40 hours per week"), or define relationships among entities ("employee X is supervised by employee Y if Y can authorize a change to X's salary").

Note that none of these requirements specify how the system is to be implemented. There is no mention of what database management system to use, whether a client-server architecture will be employed, how much memory the computer is to have, or what programming language must be used to develop the system. These implementation-specific descriptions are not considered to be requirements (unless they are mandated by the customer). The goal of the requirements phase is to understand the customer's problems and needs. Thus, requirements focus on the customer and the problem, not on the solution or the implementation. We often say that requirements designate *what* behavior the customer wants, without saying *how* that behavior will be realized. Any discussion of a solution is premature until the problem is clearly defined.

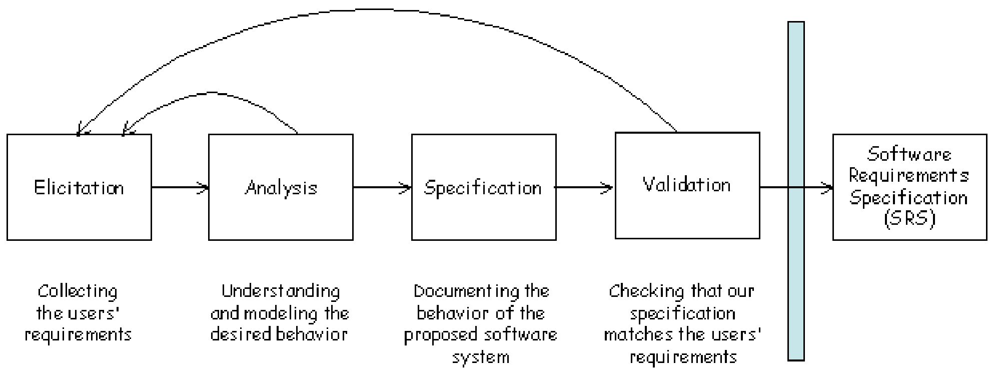

Elicitation
Analysis
Specification
Validation
Software Requirements Specification (SRS)
Collecting the users' requirements
Understanding and modeling the desired behavior
Documenting the behavior of the proposed software system
Checking that our specification matches the users' requirements

FIGURE 4.1 Process for capturing the requirements.

It helps to describe requirements as interactions among real-world phenomena, without any reference to system phenomena. For example, billing requirements should refer to customers, services billed, billing periods and amounts – without mentioning system data or procedures. We take this approach partly to get at the heart of the customer's needs; sometimes the stated needs are not the real needs. Moreover, the customer's problem is usually most easily stated in terms of the customer's business. Another reason we take this approach is to give the designer maximum flexibility in deciding how to carry out the requirements. During the **specification** phase, we will decide which requirements will be fulfilled by our software system (as opposed to requirements that are addressed by special-purpose hardware devices, by other software systems, or by human operators or users); during the **design** phase, we will devise a plan for how the specified behavior will be implemented.

Figure 4.1 illustrates the process of determining the requirements for a proposed software system. The person performing these tasks usually goes by the title of **requirements analyst** or **systems analyst**. As a requirements analyst, we first work with our customers to elicit the requirements, by asking questions, examining current behavior, or demonstrating similar systems. Next, we capture the requirements in a model or a prototype. This exercise helps us to better understand the required behavior, and usually raises additional questions about what the customer wants to happen in certain situations (e.g., what if an employee leaves the company in the middle of a pay period?). Once the requirements are well understood, we progress to the specification phase, in which we decide which parts of the required behavior will be implemented in software. During validation, we check that our specification matches what the customer expects to see in the final product. Analysis and validation activities may expose problems or omissions in our models or specification that cause us to revisit the customer and revise our models and specification. The eventual outcome of the requirements process is a Software Requirements Specification (SRS), which is used to communicate to other software developers (designers, testers, maintainers) how the final product ought to behave. Sidebar 4.2 discusses how the use of agile methods affects the requirements process and the resulting requirements documents. The remainder of this chapter explores the requirements process in more detail.

SIDEBAR 4.2 AGILE REQUIREMENTS MODELING

As we noted in Chapter 2, requirements analysis plays a large role in deciding whether to use agile methods as the basis for software development. If the requirements are tightly coupled and complex, or if future requirements and enhancements are likely to cause major changes to the system's architecture, then we may be better off with a "heavy" process that emphasizes up-front modeling. In a heavy process, developers put off coding until, the requirements have been modeled and analyzed, an architecture is proposed that reflects the requirements, and a detailed design has been chosen. Each of these steps requires a model, and the models are related and coordinated so that the design fully implements the requirements. This approach is most appropriate for large-team development, where the documentation helps the developers to coordinate their work, and for safety-critical systems, where a system's correctness and safety are more important that its release date.

However, for problems where the requirements are uncertain, it can be cumbersome to employ a heavy process and have to update the models with every change to the requirements. As an alternative approach, agile methods gather and implement the requirements in increments. The initial release implements the most essential requirements, as defined by the stakeholders' business goals. As new requirements emerge, with use of the system or with better understanding of the problem, they are implemented in subsequent releases of the system. This incremental development allows for "early and continuous delivery of valuable software" (Beck et al. 2001) and accommodates emergent and late-breaking requirements.

Extreme programming (XP) takes agile requirements processes to the extreme, in that the system is built to the requirements that happen to be defined at the time, with no planning or designing for possible future requirements. Moreover, XP forgoes traditional requirements documentation, and instead encodes the requirements as test cases that the eventual implementation must pass. Berry (2002a) points out that the trade-off for agile methods' flexibility is the difficulty of making changes to the system as requirements are added, deleted or changed. But there can be additional problems: Because XP uses test cases to specify requirements, a poorly-written test case can lead to the kinds of misunderstandings described in this chapter.

4.2 REQUIREMENTS ELICITATION

Requirements elicitation is an especially critical part of the process. We must use a variety of techniques to determine what the users and customers really want. Sometimes, we are automating a manual system, so it is easy to examine what is already done. But often, we must work with users and customers to understand a completely new problem. This task is rarely as simple as asking the right questions to pluck the requirements from the customer's head. At the early stage of a project, requirements are ill-formed and ill-understood by everyone. Customers are not always good at describing exactly what they want or need, and we are not always good at understanding someone else's business concerns. The customers know their business, but they cannot always describe their business problems to outsiders; their descriptions are full of jargon and assumptions with which we may not be familiar. Likewise, we as developers know about computer solutions, but not always about how possible solutions will affect our customers' business activities. We, too, have our jargon and assumptions, and sometimes we think everyone is speaking the same language, when in fact people have different meanings for the same words. It is only by discussing the requirements with everyone who has a stake in the system, coalescing these different views into a coherent set of requirements, and reviewing these documents with the stakeholders that we all come to an agreement about what the requirements are. (See Sidebar 4.3 for an alternative viewpoint.) If we cannot agree on what the requirements are, then the project is doomed to fail.

So who are the stakeholders? It turns out that there are many people who have something to contribute to the requirements of a new system:

- *Clients, who are the ones paying for the software to be developed:* By paying for the development, the clients are, is some sense, the ultimate stakeholders, and have final say in what the product does (Robertson and Robertson 1999).
- *Customers, who buy the software after it is developed:* Sometimes the customer and the user are the same; other times, the customer is a business manager who is interested in improving the productivity of her employees. We have to understand the customers' needs well enough to build a product that they will buy and find useful.
- *Users, who are familiar with the current system and will use the future system:* These are the experts on how the current system works, which features are the most useful, and which aspects of the system need improving. We may want to consult also with special-interest groups of users, such as users with disabilities, people who are unfamiliar with or uncomfortable using computers, expert users, and so on, to understand their particular needs.
- *Domain experts, who are familiar with the problem that the software must automate:* For example, we would consult a financial expert if we were building a financial package, or a meteorologist if our software were to model the weather. These people can contribute to the requirements, or will know about the kinds of environments to which the product will be exposed.
- *Market researchers, who have conducted surveys to determine future trends and potential customers' needs:* They may assume the role of the customer if our software is being developed for the mass market, and no particular customer has been identified yet.

SIDEBAR 4.3 USING VIEWPOINTS TO MANAGE INCONSISTENCY

Although most software engineers strive for consistent requirements, Easterbrook and Nuseibeh (1996) argue that it is often desirable to tolerate and even encourage inconsistency during the requirements process. They claim that because the stakeholders' understanding of the domain and their needs evolve over time, it is pointless to try to resolve inconsistencies early in the requirements process. Early resolutions are expensive and often unnecessary (and can occur naturally, as stakeholders revise their views). They can also be counter-productive, if the resolution process focuses attention on how to come to agreement rather than on the underlying causes of the inconsistency (e.g., stakeholders' misunderstanding of the domain).

Instead, Easterbrook and Nuseibeh propose that stakesholders' views be documented and maintained as separate Viewpoints (Nuseibeh et al. 1994) throughout the software development process. The requirements analyst defines consistency rules that should apply between Viewpoints (e.g., how objects, states, or transitions in one Viewpoint correspond to similar entities in another Viewpoint; or how one Viewpoint refines another Viewpoint), and the Viewpoints are analyzed (possibly automatically) to see if they conform to the consistency rules. If the rules are violated, the inconsistencies are recorded as part of the Viewpoints, so that other software developers do not mistakenly implement a view that is being contested. The recorded inconsistencies are rechecked whenever an associated Viewpoint is modified, to see if the Viewpoints are still inconsistent; and the consistency rules are checked periodically, to see if any have been broken by evolving Viewpoints.

The outcome of this approach is a requirements document that accommodates all stakeholders' views at all times. Inconsistencies are highlighted but not addressed until there is sufficient information to make an informed decision. This way, we avoid committing ourselves prematurely to requirements or design decisions.

- *Lawyers or auditors, who are familiar with government, safety, or legal requirements:* For example, we might consult a tax expert to ensure that a payroll package adheres to the tax law. We may also consult with experts on standards that are relevant to the product's functions.
- *Software engineers or other technology experts:* These experts ensure that the product is technically and economically feasible. They can educate the customer about innovative hardware and software technologies, and can recommend new functionality that takes advantage of these technologies. They can also estimate the cost and development time of the product.

Each stakeholder has a particular view of the system and how it should work, and often these views conflict. One of the many skills of a requirements analyst is the ability to understand each view and capture the requirements in a way that reflects the concerns of each participant. For example, a customer may specify that a system perform a particular task, but the customer is not necessarily the user of the proposed system. The user may request that the task be performed in three modes: a learning mode, a novice mode, and an expert mode; this separation will allow the user to learn and master the system gradually. Many word processing systems are implemented in this way, so that new users can adapt to the new system gradually. However, conflicts can arise when ease of use suggests a slower system than response-time requirements permit.

Also, different participants may expect differing levels of detail in the requirements documentation, in which case the requirements will need to be packaged in different ways for different people. In addition, users and developers may have preconceptions (right or wrong) about what the other group values and how it acts. Table 4.1 summarizes some of the common stereotypes. This table emphasizes the role that human interaction plays in the development of software systems; good requirements analysis requires excellent interpersonal skills as well as solid technical skills. The book's web site contains suggestions for addressing each of these differences in perception.

In addition to interviewing stakeholders, other means of eliciting requirements include

- Reviewing available documentation, such as documented procedures of manual tasks, and specifications or user manuals of automated systems
- Observing the current system (if one exists), to gather objective information about how the users perform their tasks, and to better understand the system we are about to automate or to change; often, when a new computer system is developed, the old system continues to be used because it provides some critical function that the designers of the new system overlooked
- Apprenticing with users (Beyer and Holtzblatt 1995), to learn about users' tasks in more detail, as the user performs them

TABLE 4.1 How Users and Developers View Each Other (Scharer 1990)

How developers see users	How users see developers
Users don't know what they want.	Developers don't understand operational needs.
Users can't articulate what they want.	Developers can't translate clearly stated needs into a successful system.
Users are unable to provide a usable statement of needs.	Developers set unrealistic standards for requirements definition.
Users have too many needs that are politically motivated.	Developers place too much emphasis on technicalities .
Users want everything right now.	Developers are always late.
Users can't remain on schedule.	Developers can't respond quickly to legitimately changing needs.
Users can't prioritize needs.	Developers are always over budget.
Users are unwilling to compromise.	Developers say no all the time.
Users refuse to take responsibility for the system.	Developers try to tell us how to do our jobs.
Users are not committed to development projects.	Developers ask users for time and effort, even to the detriment of the users' important primary duties.

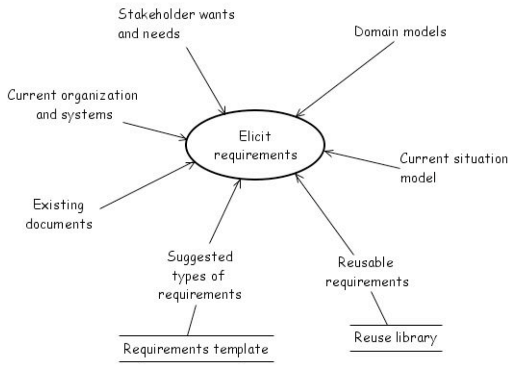


FIGURE 4.2 Sources of possible requirements (Robertson and Robertson 1999).

- Interviewing users or stakeholders in groups, so that they will be inspired by each others' ideas
- Using domain-specific strategies, such as Joint Application Design (Wood and Silver 1995) or PIECES (Wetherbe 1984) for information systems, to ensure that stakeholders consider specific types of requirements that are relevant to their particular situations
- Brainstorming with current and potential users about how to improve the proposed product

The Volere requirements process model (Robertson and Robertson 1999), as shown in Figure 4.2, suggests some additional sources for requirements, such as templates and libraries of requirements from related systems that we have developed.

4.3 TYPES OF REQUIREMENTS

When most people think about requirements, they think about required *functionality*: What services should be provided? What operations should be performed? What should be the reaction to certain stimuli? How does required behavior change over time and in response to the history of events? A **functional requirement** describes required behavior in terms of required activities, such as reactions to inputs, and the state of each entity before and after an activity occurs. For instance, for a payroll system, the functional requirements state how often paychecks are issued, what input is necessary for a paycheck to be printed, under what conditions the amount of pay can be changed, and what causes the removal of an employee from the payroll list.

The functional requirements define the boundaries of the solution space for our problem. The solution space is the set of possible ways that software can be designed to implement the requirements, and initially that set can be very large. However, in practice it is usually not enough for a software product to compute correct outputs; there are other types of requirements that also distinguish between acceptable and unacceptable products. A **quality requirement,** or **nonfunctional requirement,** describes some quality characteristic that the software solution must possess, such as fast response time, ease of use, high reliability, or low maintenance costs. A **design constraint** is a design decision, such as choice of platform or interface components, that has already been made and that restricts the set of solutions to our problem. A **process constraint** is a restriction on the techniques or resources that can be used to build the system. For example, a customer may insist that we use agile methods, so that they can use early versions of the system while we continue to add features. Table 4.2 gives examples of each kind of requirement. Thus, quality requirements, design constraints, and process constraints further restrict our solution space by differentiating acceptable, well-liked solutions from unused products.

Quality requirements sometimes sound like motherhood characteristics that all products ought to possess. After all, who is going to ask for a slow, unfriendly, unreliable, unmaintainable software system? It is better to think of quality requirements as design criteria that can be optimized and can be used to choose among alternative implementations of functional requirements. Given this approach, the question to be answered by the requirements is: To what extent must a product satisfy these quality requirements to be acceptable? Sidebar 4.4 explains how to express quality requirements so that we can test whether the requirements are met.

TABLE 4.2 Questions to tease out different types of requirements

Functional Requirements

Functionality

- What will the system do?
- When will the system do it?
- Are there several modes of operation?
- What kinds of computations or data transformations must be performed?
- What are the appropriate reactions to possible stimuli?

Data

- For both input and output, what should the format of the data be?
- Must any data be retained for any period of time?

Design Constraints

Physical Environment

- Where is the equipment to be located?
- Is there one location or several?
- Are there any environmental restrictions, such as temperature, humidity, or magnetic interference?
- Are there any constraints on the size of the system?
- Are there any constraints on power, heating, or air conditioning?
- Are there constraints on the programming language because of existing software components?

Interfaces

- Is input coming from one or more other systems?
- Is output going to one or more other systems?
- Is there a prescribed way in which input/output data must be formatted?
- Is there a prescribed medium that the data must use?

Users

- Who will use the system?
- Will there be several types of users?
- What is the skill level of each user?

Process Constraints

Resources

- What materials, personnel, or other resources are needed to build the system?
- What skills must the developers have?

Documentation

- How much documentation is required?
- Should it be on-line, in book format, or both?
- To what audience should each type of documentation be addressed?

Standards

Quality Requirements

Performance

- Are there constraints on execution speed, response time, or throughput?
- What efficiency measures will apply to resource usage and response time?
- How much data will flow through the system?
- How often will data be received or sent?

Usability and Human Factors

- What kind of training will be required for each type of user?
- How easy should it be for a user to understand and use the system?
- How difficult should it be for a user to misuse the system?

Security

- Must access to the system or information be controlled?
- Should user's data be isolated from others?
- Should user programs be isolated from other programs and from the operating system?
- Should precautions be taken against theft or vandalism?

Reliability and Availability

- Must the system detect and isolate faults?
- What is the prescribed mean time between failures?
- Is there a maximum time allowed for restarting the system after a failure?
- How often will the system be backed up?
- Must backup copies be stored at a different location?
- Should precautions be taken against fire or water damage?

Maintainability

- Will maintenance merely correct errors or will it also include improving the system?
- When and in what ways might the system be changed in the future?
- How easy should it be to add features to the system?
- How easy should it be to port the system from one platform (computer, operating system) to another?
- What materials, personnel, or other resources are needed to use and maintain the system?

Precision and Accuracy

- How accurate must data calculations be?
- To what degree of precision must calculations be made?

Time to Delivery / Cost

- Is there a prescribed timetable for development?
- Is there a limit on the amount of money to be spent on development or on hardware or software

SIDEBAR 4.4 MAKING REQUIREMENTS TESTABLE

In writing about good design, Alexander (1979) encourages us to make our requirements testable. By this, he means that once a requirement is stated, we should be able to determine whether or not a proposed solution meets the requirement. This evaluation must be objective; that is, the conclusion as to whether the requirement is satisfied must not vary according to who is doing the evaluation.

Robertson and Robertson (1997) point out that testability (which they call "measurability") can be addressed when requirements are being elicited. The idea is to quantify the extent to which each requirement must be met. These **fit criteria** form objective standards for judging whether a proposed solution satisfies the requirements. When such criteria cannot be easily expressed, then the requirement is likely to be ambiguous, incomplete, or incorrect. For example, a customer may state a quality requirement this way:

Water quality information must be accessible immediately.

How do you test that your product meets this requirement? The customer probably has a clear idea about what "immediately" means, and that notion must be captured in the requirement. We can restate more precisely what we mean by "immediately":

Water quality records must be retrieved within 5 seconds of a request.

This second formulation of the requirement can be tested objectively: A series of requests is made, and we check that system supplies the appropriate record within 5 seconds of each request.

It is relatively easy to determine fit criteria for quality requirements that are naturally quantitative (e.g., performance, size, precision, accuracy, time to delivery). What about more subjective quality requirements, like usability or maintainability? In these cases, developers have used focus groups or metrics to evaluate fit criteria:

- *75% of users shall judge the new system to be as usable as the existing system.*
- *After training, 90% of users shall be able to process a new account within 5 minutes.*
- *A module will encapsulate the data representation of at most one data type.*
- *Computation errors shall be fixed within 3 weeks of being reported.*

Fit criteria that cannot not be evaluated before the final product is delivered are harder to assess:

- *The system shall not be unavailable for more than a total maximum of 3 minutes each year.*
- *The mean-time-between-failures shall be no more than 1 year.*

In these cases, we either estimate a system's quality attributes (e.g., there are techniques for estimating system reliability, and for estimating the number of bugs per lines of code) or evaluate the delivered system during its operation – and suffer some financial penalty if the system does not live up to its promise.

Interestingly, what gets measured gets done. That is, unless a fit criterion is unrealistic, it will probably be met. The key is to determine, with the customer, just how to demonstrate that a delivered system meets its requirements. The Robertsons suggest three ways to help make requirements testable:

- Specify a quantitative description for each adverb and adjective so that the meaning of qualifiers is clear and unambiguous.
- Replace pronouns with specific names of entities.
- Make sure that every noun is defined in exactly one place in the requirements documents.

An alternative approach, advocated by the Quality Function Deployment (QFD) school (Akao 1990), is to realize quality requirements as special-purpose functional requirements, and to test quality requirements by testing how well their associated functional requirements have been satisfied. This approach works better for some quality requirements than it does for others. For example, real-time requirements can be expressed as additional conditions or constraints on when required functionality occurs. Other quality requirements, such as security, maintainability, and performance, may be satisfied by adopting existing designs and protocols that have been developed specifically to optimize a particular quality requirement.

Resolving Conflicts

In trying to elicit all types of requirements from all of the relevant stakeholders, we are bound to encounter conflicting ideas of what the requirements ought to be. It usually helps to ask the customer to prioritize requirements. This task forces the customer to reflect on which of her requested services and features are most essential. A loose prioritization scheme might separate requirements into three categories:

1. Requirements that absolutely must be met *(Essential)*
2. Requirements that are highly desirable but not necessary *(Desirable)*
3. Requirements that are possible but could be eliminated *(Optional)*

For example, a credit card billing system must be able to list current charges, sum them, and request payment by a certain date; these are *essential* requirements. But the billing system may also separate the charges by purchase type, to assist the purchaser in understanding buying patterns. Such purchase-type analysis is a *desirable,* but probably a nonessential requirement. Finally, the system may print the credits in black and the debits in red, which would be useful but is *optional*. Prioritizing requirements by category is helpful to all parties in understanding what is really needed. It is also useful when a software development project is constrained by time or resources; if the system as defined will cost too much or take too long to develop, optional requirements can be dropped, and desirable requirements can be analyzed for elimination or postponement to later versions.

Prioritization can be especially helpful in resolving conflicts among quality requirements; often, two quality attributes will conflict, so that it is impossible to optimize for both. For example, suppose a system is required to be maintainable and deliver responses quickly. A design that emphasizes maintainability through separation of concerns and encapsulation may slow the performance. Likewise, tweaking a system to perform especially well on one platform affects its portability to other platforms, and secure systems necessarily control access and restrict availability to some users. Emphasizing security, reliability, robustness, usability, or performance can all affect maintainability, in that realizing any of these characteristics increases the design's complexity and decreases its coherence. Prioritizing quality requirements forces the customer to choose those software quality factors about which she cares most – which helps us to provide a reasonable, if not optimal, solution to the customer's quality requirements.

We can also avoid trying to optimize multiple conflicting quality requirements by identifying and aiming to achieve *fit criteria*, which establish clear-cut acceptance tests for these requirements (see Sidebar 4.4). But what if we cannot satisfy these fit criteria? Then it may be time to reevaluate the stakeholders' views and to employ negotiation. However, negotiation is not easy; it requires skill, patience and experience in finding mutually acceptable solutions. Fortunately, stakeholders rarely disagree about the underlying problem that the software system is addressing. More likely, conflicts will pertain to possible approaches to, or design constraints on, solving the problem (e.g., stakeholders may insist on using different database systems, different encryption algorithms, different user interfaces, or different programming languages). More seriously, stakeholders may disagree over priorities of requirements, or about the business policies to be incorporated into the system. For example, a university's colleges or departments may want different policies for evaluating students in their respective programs, whereas university administrators may prefer consolidation and uniformity. Resolution requires determining exactly why each stakeholder is adamant about a particular approach, policy, or priority ranking – for example, they may be concerned about cost, security, speed, or quality – and the we need to work towards agreement on fundamental requirements. With effective negotiation, the stakeholders will come to understand and appreciate each other's fundamental needs, and will strive for a resolution that satisfies everyone; such resolutions are usually very different from any of the stakeholders' original views.

Two Kinds of Requirements Documents

In the end, the requirements are used by many different people and for different purposes. Requirements analysts and their clients use requirements to explain their understanding of how the system should behave. Designers treat requirements as constraints on what would be considered an acceptable solution. The test team derives from the requirements a suite of *acceptance tests,* which will be used to demonstrate to the customer that the system being delivered is indeed what was ordered. The maintenance team uses the requirements to help ensure that system enhancements (bug fixes, new features) do not interfere with the system's original intent. Sometimes a single document can serve all of these needs, leading to a common understanding among customers, requirements analysts, and developers. But often two documents are needed: a *requirements definition* that is aimed at a business audience, such as clients, customers, users, and a *requirements specification* that is aimed at a technical audience, such as designers, testers, project managers.

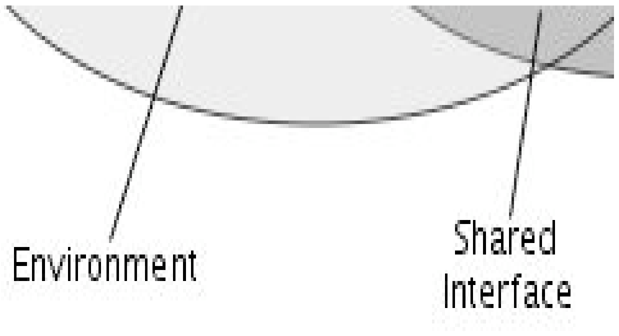


FIGURE 4.3 Requirements vs. Specification.

Zave and Jackson (1997) distinguish between these two types of documents; we illustrate the distinction using a small running example from Jackson and Zave (1995). Consider a software-controlled turnstile situated at the entrance to a zoo. When the turnstile is fed a coin, it unlocks, allowing a visitor to push his way through the turnstile and enter the zoo. Once an unlocked turnstile has rotated enough to allow one entry, the turnstile locks again, to prevent another person from entering without payment.

A **requirements definition** is a complete listing of everything the customer wants to achieve. The document expresses requirements by describing the entities in the environment in which the proposed system will be installed, and by describing the customer's desired constraints on, monitoring of, or transformations of those entities. The purpose of the proposed system is to realize these requirements (Zave and Jackson 1997). Thus, the requirements are written entirely in terms of the environment, describing how the environment will be affected by the proposed system. Our turnstile example has two requirements: (1) no one should enter the zoo without paying an entrance fee, and (2) for every entrance fee paid, the system should not prevent a corresponding entry.[1] The requirements definition is typically written jointly by the client and the requirements analyst, and it represents a contract describing what functionality the developer promises to deliver to the client.

The **requirements specification** restates the *requirements* as a *specification* of how the proposed system shall behave. The specification also is written entirely in terms of the environment – except that it refers solely to environmental entities that are accessible to the system via its interface; that is, the system boundary makes explicit those environmental entities that can be monitored or controlled by the system. This distinction is depicted in Figure 4.3, with requirements defined anywhere within the environment's domain, including, possibly, the system's interface, and with the specification restricted only to the intersection between the environment and system domains. To see the distinction, consider the requirement that no one should enter the zoo without paying an entrance fee. If the turnstile has a coin slot and is able to detect when a valid coin is inserted, then it can determine when an entrance fee is being paid. In contrast, the concept of an entry event may be outside the scope of the system.Thus, the requirement must be

[1] A more intuitive expression of this second requirement, that anyone who pays should be allowed to enter the zoo, is not implementable. There is no way for the system to prevent external factors from keeping the paid visitor from entering the zoo: another visitor may push his way through the unlocked turnstile before the paid visitor, the zoo may close before the paid visitor enters the turnstile, the paid visitor may change his mind and leave, and so on (Jackson and Zave 1995).

rewritten to realize entry events using only events and states that the turnstile *can* detect and control, such as whether the turnstile is unlocked and whether it detects a visitor pushing the turnstile:

> *When a visitor applies a certain amount of force on an unlocked turnstile, the turnstile will automatically rotate a one-half turn, ending in a locked position.*

In this way, the specification refines the original requirements definition.

The requirements specification is written by the requirements analyst, and is used by the other software developers. The analyst must be especially careful that no information is lost or changed when refining the requirements into a specification. There must be a direct correspondence between each requirement in the definition document and those in the specification document.

4.4 CHARACTERISTICS OF REQUIREMENTS

To ensure that the eventual product is successful, it is important that the requirements be of high quality; what isn't specified usually isn't built. We discuss later in this chapter how to validate and verify requirements. In the meantime, we list below the desirable characteristics for which we should check.

1. *Are the requirements correct?* Both we and the customer should review the documented requirements, to ensure that they conform to our understanding of the requirements.
2. *Are the requirements consistent?* That is, are there no conflicting requirements? For example, if one requirement states that a maximum of 10 users can be using the system at one time, and another requirement says that in a certain situation there may be 20 simultaneous users, the two requirements are said to be inconsistent. In general, two requirements are **inconsistent** if it is impossible to satisfy both simultaneously.
3. *Are the requirements unambiguous?* The requirements are ambiguous if multiple readers of the requirements can walk away with different but valid interpretations. Suppose a customer for a satellite control system requires the accuracy to be sufficient to support mission planning. The requirement does not tell us what mission planning requires for support. The customer and the developers may have very different ideas as to what level of accuracy is needed. Further discussion of the meaning of "mission planning" may result in a more precise requirement: "In identifying the position of the satellite, position error shall be less then 50 feet along orbit, less than 30 feet off orbit." Given this more detailed requirement, we can test for position error and know exactly whether or not we have met the requirement.
4. *Are the requirements complete?* The set of requirements is **complete** if it specifies required behavior and output for all possible inputs in all possible states under all possible constraints. Thus, a payroll system should describe what happens when an employee takes a leave without pay, gets a raise, or needs an advance. We say that the requirements are **externally complete** if all states, state changes, inputs, products, and constraints are described by some requirement. A requirements description is **internally complete** if there are no undefined terms among the requirements.
5. *Are the requirements feasible?* That is, does a solution to the customer's needs even exist? For example, suppose the customer wants users to be able to access a main computer that is located several thousand miles away and have the response time for remote users be the same as for local users (whose workstations are connected directly to the main computer). Questions of feasibility often arise when the customer requires two or more quality requirements, such as a request for an inexpensive system that analyzes huge amounts of data and outputs the analysis results within seconds.
6. *Is every requirement relevant?* Sometimes a requirement restricts the developers unnecessarily, or includes functions that are not directly related to the customer's needs. For example, a general may decide that a tank's new software system should allow soldiers to send and receive electronic mail, even though the main purpose of the tank is to traverse uneven terrain. We should endeavor to keep

this "feature explosion" under control, and to help keep stakeholders focused on their essential and desirable requirements.

7. *Are the requirements testable?* The requirements are **testable** if they suggest acceptance tests that would clearly demonstrate whether the eventual product meets the requirements. Consider how we might test the requirement that a system provide real-time response to queries. We do not know what "real-time response" is. However, if fit criteria were given, saying that the system shall respond to queries in not more than two seconds, then we know exactly how to test the system's reaction to queries.

8. *Are the requirements traceable?* Are the requirements organized and uniquely labeled for easy reference? Does every entry in the requirements definition have corresponding entries in the requirements specification, and vice versa?

We can think of these characteristics as the functional and quality requirements for a set of product requirements. These characteristics can help us to decide when we have collected enough information, and when we need to learn more about what a particular requirement means. As such, the degree to which we want to satisfy these characteristics will affect the type of information that we gather during requirements elicitation, and how comprehensive we want to be. It will also affect the specification languages we choose to express the requirements and the validation and verification checks that we eventually perform to access the requirements.

4.5 MODELING NOTATIONS

One trait of an engineering discipline is that it has repeatable processes, such as the techniques presented in Chapter 2, for developing safe and successful products. A second trait is that there exist standard notations for modeling, documenting, and communicating decisions. Modeling can help us to understand the requirements thoroughly, by teasing out what questions we should be asking. Holes in our models reveal unknown or ambiguous behavior. Multiple, conflicting outputs to the same input reveal inconsistencies in the requirements. As the model develops, it becomes more and more obvious what we don't know and what the customer doesn't know. We cannot complete a model without understanding the subject of the model. Also, by restating the requirements in a completely different form from the customer's original requests, we force the customer to examine our models carefully in order to validate the models' accuracy.

If we look at the literature, we see that there are a seemingly infinite number of specification and design notations and methods, and that new notations are being introduced and marketed all the time. But if we step back and ignore the details, we see that many notations have a similar look and feel. Despite the number of individual notations, there are probably fewer than ten basic paradigms for expressing information about a problem's concepts, behavior, and properties.

This section focuses on seven basic notational paradigms that can be applied in several ways to steps in the development process. We begin our discussion of each by first introducing the paradigm and the types of problems and descriptions for which it is particularly apt. Then we describe one or two concrete examples of notations from that paradigm. Once you are familiar with the paradigms, you can easily learn and use a new notation because you will understand how it relates to existing notations.

However, caution is advised. We need to be especially careful about the terminology we use when modeling requirements. Many of the requirements notations are based on successful design methods and notations, which means that most other references for the notations provide examples of designs, rather than of requirements, and give advice about how to make design-oriented modeling decisions. Requirements decisions are made for different reasons, so the terminology is interpreted differently. For example, in requirements modeling we discuss *decomposition, abstraction,* and *separation of concerns* – all of which were originally design techniques for creating elegant, modular designs. We decompose a requirements specification along separate concerns to simplify the resulting model and make it easier to read and to understand. In contrast, we decompose a design to improve the system's quality attributes (modularity, maintainability, performance, time to delivery, etc.); the requirements name and constrain those attributes, but decomposition plays no role in this aspect of specification. Thus, although we use the terms *decomposition* and *modularity* in both specification and design, the decomposition decisions we make at each stage are different because they have different goals.

Throughout this section, we illustrate notations by using them to model aspects of the turnstile problem introduced earlier (Zave and Jackson 1997), and a library problem. The library needs to track its texts and other materials, its loan records, and information about its patrons. Popular items are placed on *reserve*, meaning that their loan periods are shorter than those of other books and materials, and that the penalty for returning them late is higher than the late penalty for returning unreserved items.

Entity-Relationship Diagrams

Early in the requirements phase, it is convenient to build a conceptual model of the problem that identifies what objects or entities are involved, what they look like (by defining their attributes), and how they relate to one another. Such a model designates names for the basic elements of the problem. These elements are then reused in other descriptions of the requirements (possibly written in other notations) that specify how the objects, their attributes, and their relationships would change in the course of the proposed system's execution. Thus, the conceptual model helps to tie together multiple views and descriptions of the requirements.

The **Entity-Relationship diagram (ER diagram)** (Chen 1976) is a popular, graphical notational paradigm for representing conceptual models. As we will see in Chapter 6, it forms the basis of most object-oriented requirements and design notations, where it is used to model the relationships among objects in a problem description, or to model the structure of a software application. This notation paradigm is also popular for describing database schema (i.e., describing the logical structure of data stored in a database).

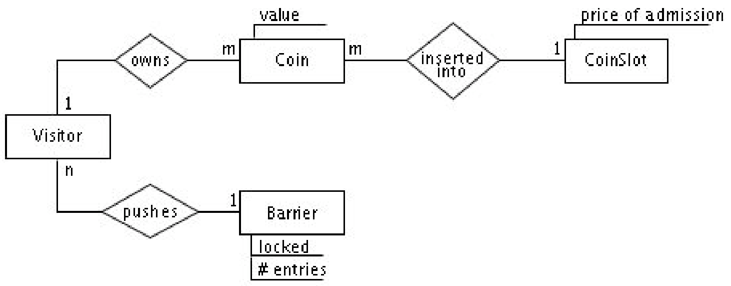


FIGURE 4.4 Entity-relationship diagram of turnstile problem.

ER diagrams have three core constructs – entities, attributes, and relations – that are combined to specify a problem's elements and their interrelationships. Figure 4.4 is an ER diagram of the turnstile. An **entity**, depicted as a rectangle, represents a collection (sometimes called a class) of real-world objects that have common properties and behaviors. For example, the world contains many coins, but for the purpose of modeling the turnstile problem, we treat all coins as being equivalent to one another in all aspects (such as size, shape and weight) except perhaps for their monetary value. A **relationship** is depicted as an edge between two entities, with a diamond in the middle of the edge specifying the type of relationship. An **attribute** is an annotation on an entity that describes data or properties associated with the entity. For example, in the turnstile problem, we are most interested in the coins that are inserted into the turnstile's coin slot (a relationship), and how their monetary values compare to the price for admission into the zoo (comparison of attribute values). Variant ER notations introduce additional constructs, such as attributes on relationships, 1-to-many relationships, many-to-many relationships, special relationships like inheritance, and class-based in addition to individual-entity-based attributes. For example, our turnstile model shows the cardinality (sometimes called the "arity") of the relationships, asserting that the turnstile is to admit multiple Visitors. More sophisticated notations have the concept of a **mutable entity**, whose membership or whose relations to members of other entities may change over time. For example, in an ER diagram depicting a family, family members and their interrelations change as they get married, have children, and die. By convention, the entities and relationships are laid out, so that relationships are read from left to right, or from top to bottom.

ER diagrams are popular because they provide an overview of the problem to be addressed (that is, they depict all of the parties involved), and because this view is relatively stable when changes are made to the problem's requirements. A change in requirements is more likely to be a change in how one or more entities behave than to be a change in the set of participating entities. For these two reasons, an ER diagram is likely to be used to model a problem early in the requirements process.

The simplicity of ER notations is deceiving; in fact, it is quite difficult to use ER modeling notations well in practice. It is not always obvious at what level of detail to model a particular problem, even though there are only three major language constructs. For example, should the barrier and coin slot be modeled as entities, or should they be represented by a more abstract turnstile entity? Also, it can be difficult to decide what data are entities and what data are attributes. For example, should the lock be an entity? There are arguments for and against each of these choices. The primary criteria for making decisions are whether a choice results in a clearer description, and whether a choice unnecessarily constrains design decisions.

Example: UML Class Diagrams

ER notation is often used by more complex approaches. For example, the **Unified Modeling Language (UML)** (OMG 2003) is a collection of notations used to document software specifications and designs. We will use UML extensively in Chapter 6 to describe object-oriented specifications and designs. Because UML was originally conceived for object-oriented systems, it represents systems in terms of objects and methods. Objects are akin to entities; they are organized in classes that have an inheritance hierarchy. Each object provides methods that perform actions on the object's variables. As objects execute, they send messages to invoke each other's methods, acknowledge actions, and transmit data.

The flagship model in any UML specification is the **class diagram**, a sophisticated ER diagram relating the classes (entities) in the specification. Although most UML texts treat class diagrams primarily as a design notation, it is possible and convenient to use UML class diagrams as a conceptual modeling notation, in which classes represent real-world entities in the problem to be modeled. It may be that a class in the conceptual model, such as a Customer class, corresponds to a program class in the implementation, such as a CustomerRecord, but this need not always be the case. It is the software designer's task to take a class-diagram specification and construct a suitable design model of the implementation's class structure.

In general, the kinds of real-world entities that we would want to represent in a class diagram include actors (e.g., patrons, operators, personnel); complex data to be stored, analyzed, transformed, or displayed; or records of transient events (e.g., business transactions, phone conversations). The entities in our library problem include people, like the patrons and librarians; the items in the library's inventory, like books and periodicals; and information about items out on loan, such as due dates and fines.

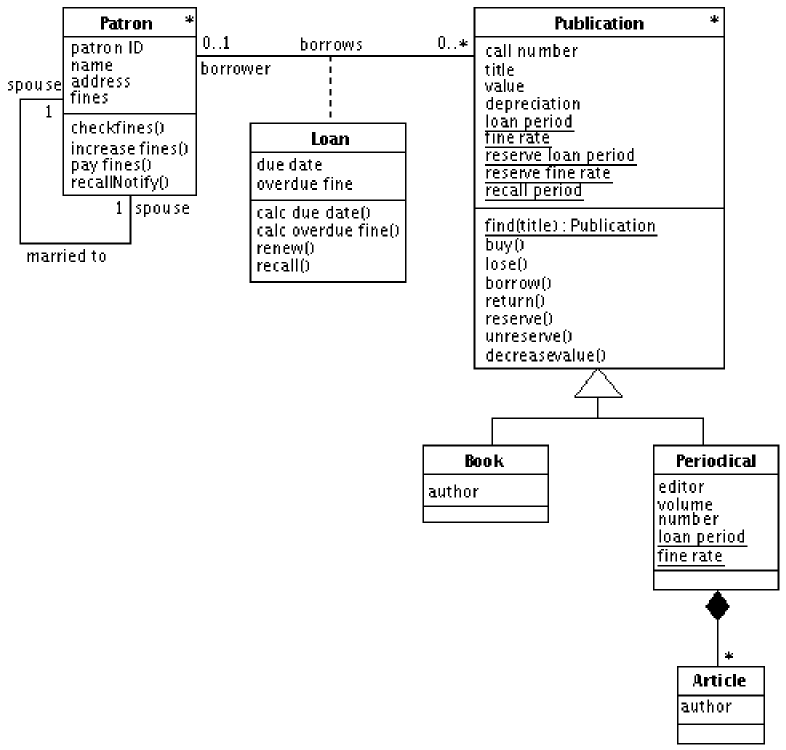


FIGURE 4.5 UML class model of the library problem.

Figure 4.5 depicts a simple UML class diagram for such a library. Each box is a **class** that represents a collection of similarly typed entities; for example, a single class represents all of the library's books. A class has a **name**; a set of **attributes**, which are simple data variables whose values can vary over time and among different entities of the class; and a set of **operations** on the class's attributes. By "simple data variable," we mean a variable whose values are too simple for the variable to be a class by itself. Thus, we model a Patron's address as an attribute, likely as one or more string values, whereas we would model a Patron's credit card information, credit institution, credit card number, expiration date, or billing address as a separate class (not shown). Note that many attributes we might expect in the library class diagram are missing (such as records and films) or are imprecisely defined (such as periodical, which doesn't distinguish between newspapers and magazines), and operations are omitted (such as dealing with book repair or loss). This imprecision is typical in early conceptual diagrams. The idea is to provide enough attributes and operations, and in sufficient detail, that anyone who reads the specification can grasp what the class represents and what its responsibilities are.

UML also allows the specifier to designate attributes and operations as being associated with the class rather than with instances of the class. A **class-scope attribute,** represented as an underlined attribute, is a data value that is shared by all instances of the class. In the library class diagram, the attributes reserve loan period and reserve fine rate are values that apply to all publications on reserve. Thus in this model, the librarian can set and modify the loan duration for classes of items (e.g., books, periodicals, items on reserve) but not for individual items. Similarly, a **class-scope operation,** written as an underlined operation, is an operation performed by the abstract class, rather than by class instances, on a new instance or on the whole collection of instances; create(),search(), and delete() are common class-scope operations.

A line between two classes, called an **association,** indicates a relationship between the classes' entities. An association may represent interactions or events that involve objects in the associated classes, such as when a Patron borrows a Publication. Alternatively, an association might relate classes in which one class is a property or element of the other class, such as the relationship between a Patron and his Credit Card. Sometimes these latter types of associations are **aggregate associations**, or "*has-a"* relationships, such as in our example. An aggregate association is drawn as an association with a white diamond on one end, where the class at the diamond end is the aggregate and it includes or owns instances of the class(es) at the other end(s) of the association. **Composition** association is a special type of aggregation, in which instances of the compound class are physically constructed from instances of the component classes (e.g., a bike consists of wheels, gears, pedals, a handlebar); it is represented as an aggregation with a black diamond. In our library model, each Periodical, such as a newspaper or magazine, is composed of Articles.

An association with a triangle on one end represents a **generalization** association, also called a *sub-type* relation or an *"is-a"* relation, where the class at the triangle end of the association is the parent class of the classes at the other ends of the association, called **subclasses**. A subclass inherits all of the parent class's attributes, operations, and associations. Thus, we do not need to specify explicitly that Patrons may borrow Books, because this association is inherited from the association between Patron and Publication. A subclass extends its inherited behavior with additional attributes, operations, and associations. In fact, a good clue as to whether we want to model an entity as a new subclass, as opposed to as an instance of an existing class, is whether we really need new attributes, operations, or associations to model the class variant. In many cases, we can model variants as class instances that have different attribute values. In our library problem, we represent whether an item is on reserve or on loan using Publication attributes[2], rather than by creating Reserved and OnLoan subclasses.

Associations can have labels, usually verbs, that describe the relationship between associated entities. An end of an association can also be labeled, to describe the role that entity plays in the association. Such **role names** are useful for specifying the context of an entity with respect to a particular association. In the library example, we might keep track of which patrons are married, so that we can warn someone when his or her spouse has overdue books. Association ends can also be annotated with **multiplicities,** which specify constraints on the number of entities and the number of links between associated entities. Multiplicities can be expressed as specific numbers, ranges of numbers, or unlimited numbers (designated "*"). A multiplicity on one end of an association indicates how many instances of that class can be linked to one instance of the associated class. Thus at any point in time, a Patron may borrow zero or more Publications, but an individual Publication can be borrowed by at most one Patron.

The Loan class in the library model is an **association class,** which relates attributes and operations to an association. Association classes are used to collect information that cannot be attributed solely to one class or another. For example, the Loan attributes are not properties of the borrower or of the item borrowed, but rather of the loan transaction or contract. An association class has exactly one instantiation per link in the association, so our modeling Loan as an association class is correct only if we want to model snapshots of the library inventory (i.e., model only current loans). If we wanted instead to maintain a history of all loan transactions, then (because a patron might borrow an item multiple times), we would model Loan as a full-fledged class.

Event Traces

Although ER diagrams are helpful in providing an overall view of the problem being modeled, the view is mostly structural, showing which entities are related; the diagram says nothing about how the entities are to behave. We need other notation paradigms for describing a system's behavioral requirements.

An **event trace** is a graphical description of a sequence of events that are exchanged between real-world entities. Each vertical line represents the time line for a distinct entity, whose name appears at the top of the line. Each horizontal line represents an event or interaction between the two entities bounding the line, usually conceived as a message passed from one entity to another. Time progresses from the top to the bottom of the trace, so if one event appears above another event, then the upper event occurs before the lower event. Each graph depicts a single **trace,** representing only one of several possible behaviors. Figure 4.6 shows two traces for the turnstile problem: the trace on the left represents typical behavior,

[2]In later examples, we model an item's loan state and reserve state as states in a state-machine model (Figure 4.9), and this information is included in the library's detailed class model (Figure 4.18).

whereas the trace on the right shows exceptional behavior of what happens when a visitor tries to sneak into the zoo by inserting a valueless token (a slug) into the coin slot.

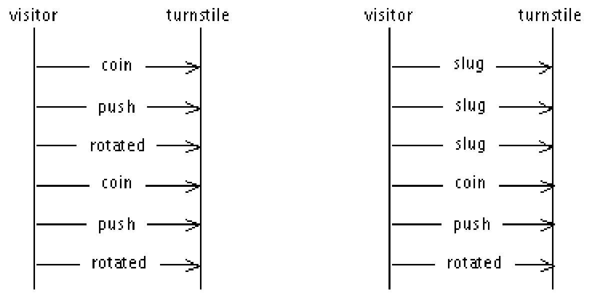


Figure 4.6 Event traces in the turnstile problem.

Event traces are popular among both developers and customers because traces have a semantics that is relatively precise, with the exception of timing issues, yet is simple and easy to understand. Much of their simplicity comes from decomposing requirements descriptions into scenarios, and considering (modeling, reading, understanding) each scenario separately as a distinct trace. Because of these same properties, event traces are not very efficient for documenting behavior. We would not want to use traces to provide a complete description of required behavior, because the number of scenarios we would have to draw can quickly become unwieldy. Instead, traces are best used at the start of a project, to come to consensus on key requirements and to help developers identify important entities in the problem being modeled.

Example: Message Sequence Chart

Message Sequence Charts (ITU 1996) are an enhanced event-trace notation, with facilities for creating and destroying entities, specifying actions and timers, and composing traces. Figure 4.7 displays an example Message Sequence Chart (MSC) for a loan transaction in our library problem. Each vertical line represents a participating **entity**, and a **message** is depicted as an arrow from the sending entity to the receiving entity; the arrow's label specifies the message name and data parameters, if any. A message arrow may slope downwards (e.g., message recall notice) to reflect the passage of time between when the message is sent and when it is received. Entities may come and go during the course of a trace; a dashed arrow, optionally annotated with data parameters, represents a *create* event that spawns a new entity, and a cross at the bottom of an entity line represents the end of that entity's execution. In contrast, a solid rectangle at the end of the line represents the end of an entity's specification without meaning the end of its execution. **Actions**, such as method invocations or changes to variable values, are specified as labeled rectangles positioned on an entity's execution line, located at the point in the trace where the action occurs. Thus, in our MSC model of a library loan, loan events are sent to the Publication being borrowed, and the Publication entity is responsible for creating a Loan entity that manages loan-specific data, such as the due date. Reserving an item that is out on loan results in a recall of that item. Returning the borrowed item terminates the loan, but not before calculating the overdue fine, if any, for returning the item after the loan's due date.

There are facilities for composing and refining Message Sequence Charts. For example, important states in an entity's evolution can be specified as **conditions**, represented as labeled hexagons. We can then specify a small collection of subtraces between conditions, and derive a variety of traces by composing the charts at points where the entities' states are the same. For example there are multiple scenarios between state publication on loan and the end of the loan transition: the patron renews the loan once, the patron renews the loan twice, the patron returns the publication, the patron reports the publication as being lost; each of these subscenarios could be appended to a prefix trace of a patron succeeding to borrow the publication. Such composition and refinement features help to reduce the number of MSCs one would need to write to specify a problem completely. However, these features do not completely address the trace-explosion problem of the event-trace paradigm, so Message Sequences Charts are usually used only to describe key scenarios, rather than to specify entire problems.

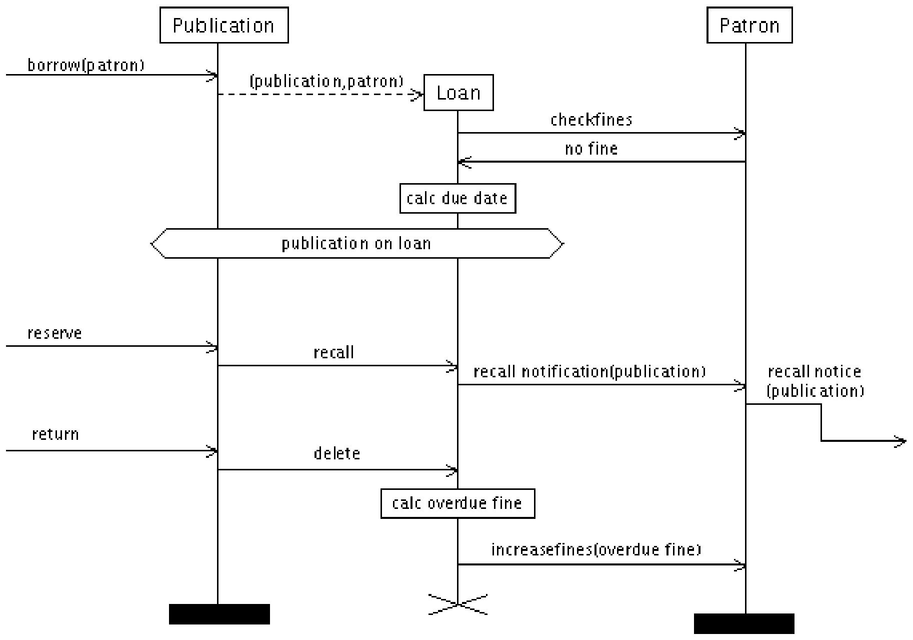


FIGURE 4.7 Message Sequence Chart for library loan transaction.

State Machines

State-machine notations are used to represent collections of event traces in a single model. A **state machine** is a graphical description of all dialog between the system and its environment. Each node, called a **state**, represents a stable set of conditions that exists between event occurrences. Each edge, called a **transition**, represents a change in behavior or condition due to the occurrence of an event; each transition is labeled with the triggering event, and possibly with an output event, preceded by the symbol "/", that is generated when the transition occurs.

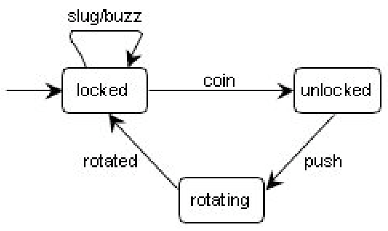


FIGURE 4.8 Finite-state machine model of the turnstile problem.

State machines are useful both for specifying dynamic behavior and for describing how behavior should change in response to the history of events that have already occurred. That is, they are particularly suited for modeling how the system's responses to the same input *change* over the course of the system's execution. For each state, the set of transitions emanating from that state designates both the set of events that can trigger a response and the corresponding responses to those events. Thus, when our turnstile (shown in Figure 4.8) is in the unlocked state, its behavior is different from when it is in state locked; in particular, it responds to different input events. If an unanticipated event occurs (for example if the user tries to push his way through the turnstile when the machine is in state locked), the event will be ignored and discarded. We could have specified this latter behavior explicitly as a transition from state locked to state locked, triggered by event push; however, the inclusion of such "no-effect" transitions can clutter the model. Thus, it is best to restrict the use of self-looping transitions to those that have an observable effect, such as an output event.

A path through the state machine, starting from the machine's initial state and following transitions from state to state, represents a trace of observable events in the environment. If the state machine is **deterministic,** meaning that for every state and event there is a unique response, then a path through the machine represents *the* event trace that will occur, given the sequence of input events that trigger the path's transitions. Example traces of our turnstile specification include

coin, push, rotated, coin, push, rotated, ...
slug, slug, slug, coin, push, rotated,...

which correspond to the event traces in Figure 4.6.

You may have encountered state machines in some of your other computing courses. In theory-of-computing courses, finite-state machines are used as automata that recognize strings in regular languages. In some sense, state-machine specifications serve a purpose similar to that of automata; they specify the sequences of input and output events that the proposed system is expected to realize. Thus, we view a state-machine specification as a compact representation of a set of desired, externally observable, event traces – just as a finite-state automaton is a compact representation of the set of strings that the automaton recognizes.

Example: UML Statechart Diagrams

A **UML statechart diagram** depicts the dynamic behavior of the objects in a UML class. A UML class diagram gives a static, big-picture view of a problem, in terms of the entities involved and their relationships; it says nothing about how the entities behave, or how their behaviors change in response to input events. A statechart diagram shows how a class's instances should change state and how their attributes should change value as the objects interact with each other. Statechart diagrams are a nice counterpart to Message Sequence Charts (MSC), in that an MSC shows the events that pass between entities without saying much about each entity's behavior, whereas a statechart diagram shows how an entity reacts to input events and generates output events.

A UML model is a collection of concurrently executing statecharts – one per instantiated object – that communicate with each other via message passing (OMG 2003). Every class in a UML class diagram has an associated statechart diagram that specifies the dynamic behavior of the objects of that class. Figure 4.9 shows the UML statechart diagram for Publication class from our Library class model.

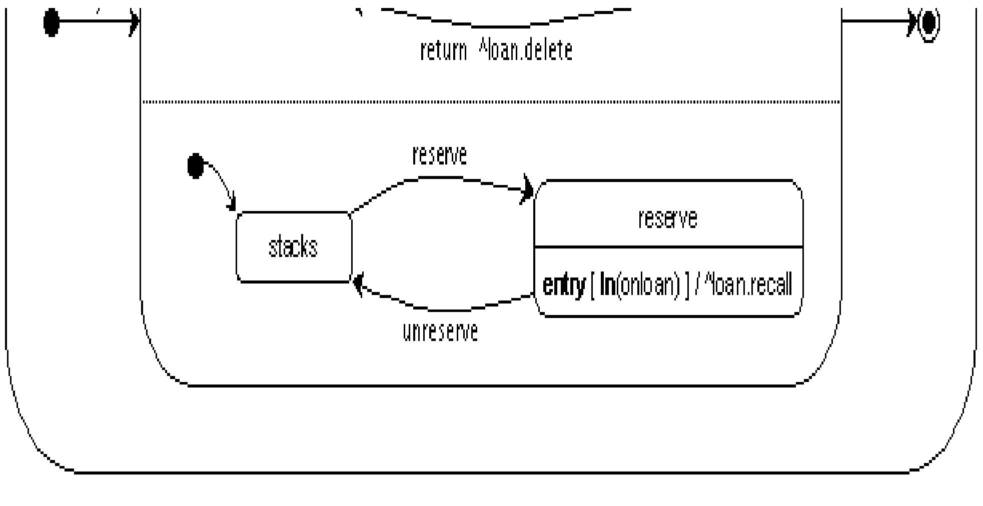


FIGURE 4.9 UML statechart diagram for the Publication class.

UML statechart diagrams have a rich syntax, much of it borrowed from Harel's original conception of Statecharts (Harel 1987), including state hierarchy, concurrency, and inter-machine communication. State hierarchy is used to unclutter diagrams by collecting into **superstates** those states with common transitions. We can think of a superstate as a submachine, with its own set of states and transitions. A transition whose destination state is a superstate acts as a transition to the superstate's default initial state, designated by an arrow from the superstate's internal black circle. A transition whose source state is a superstate acts as a set of transitions, one from each of the superstate's internal states. For example in the Publication state diagram, the transition triggered by event lost can be enabled from any of the superstate's internal states; this transition ends in a final state, designates the end of the object's life.

A superstate can actually comprise multiple concurrent submachines, separated by dashed lines. The UML statechart for Publication includes two submachines: one that indicates whether the publication is out on loan or not, and another that indicates whether the publication is on reserve or not. The submachines are said to operate **concurrently**, in that a Publication instance could at any time receive and respond to events of interest to either or both submachines. In general, concurrent submachines are used to model separate, unrelated sub-behaviors, making it easier to understand and consider each sub-behavior. An equivalent statechart for Publication in Figure 4.10 that does not make use of state hierarchy or concurrency is comparatively messy and repetitive. Note that this messy statechart has a state for each combination of states from Figure 4.9 (stacks = Publication is in library and not on reserve, onloan = Publication is on loan and not on reserve, etc.).[3]

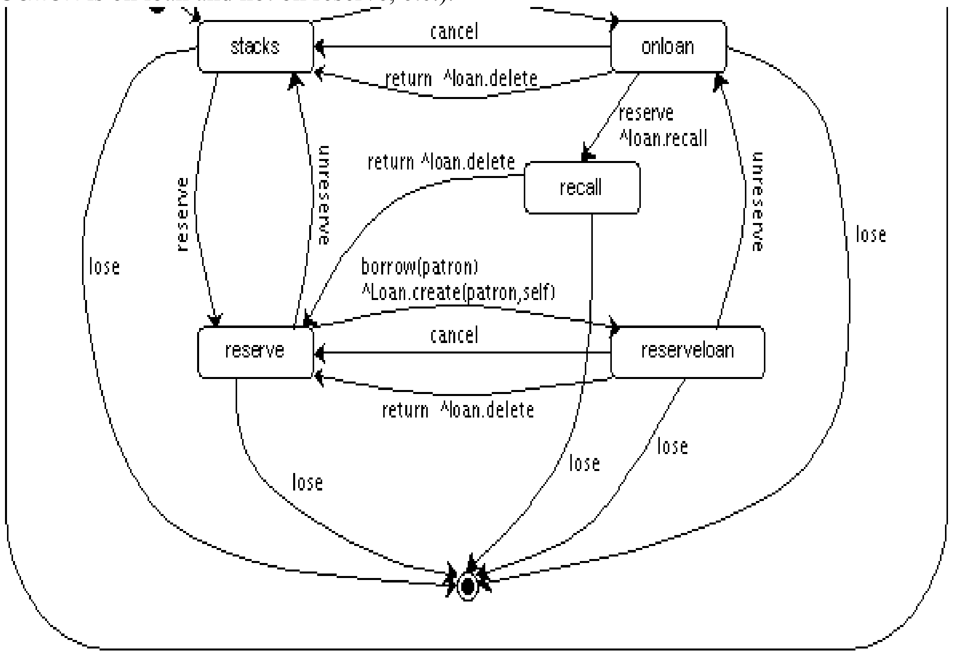


FIGURE 4.10 Messy UML statechart diagram for Publication class.

State transitions are labeled with their enabling events and conditions and with their side effects. Transition labels have syntax

event(args) [condition] /action ^Object.event(args)**

where the triggering **event** is a message from another object and which may carry parameters. The enabling **condition**, delimited by square brackets, is a predicate on the object's attribute values. If the transition is

[3] The messy statechart also has a recall state that covers the case where a publication that is being put on reserve is on loan and needs to be recalled; this behavior cannot be modeled as a transition from onloan to reserveloan, because state reserveloan has a transition cancel (used to disallow a loan request if the Patron has outstanding fines) that would be inappropriate in this situation. This special case is modeled in Figure 4.9 by testing on **entry** (keyword **entry** is explained below) to state reserve whether the concurrent submachine is **In** state loan and issuing a recall event if it is.

taken, its **actions,** each prefaced with a slash (/), specify assignments made to the object's attributes; the asterisk '*' indicates that a transition may have arbitrarily many actions. If the transition is taken, it may generate arbitrarily many **output events**, ^Object.event, each prefaced with a carat (^); an output event may carry parameters and is either designated for a target Object or is broadcast to all objects. For example, in the messy Publication statechart (Figure 4.10), the transition to state recall is enabled if the publication is in state onloan when a request to put the item on reserve is received. When the transition is taken, it sends an event to the Loan object, which in turn will notify the borrower that the item must be returned to the library sooner than the loan's due date. Each of the transition-label elements is optional. For example, a transition need not be enabled by an input event; it could be enabled only by a condition or by nothing, in which case the transition is always enabled.

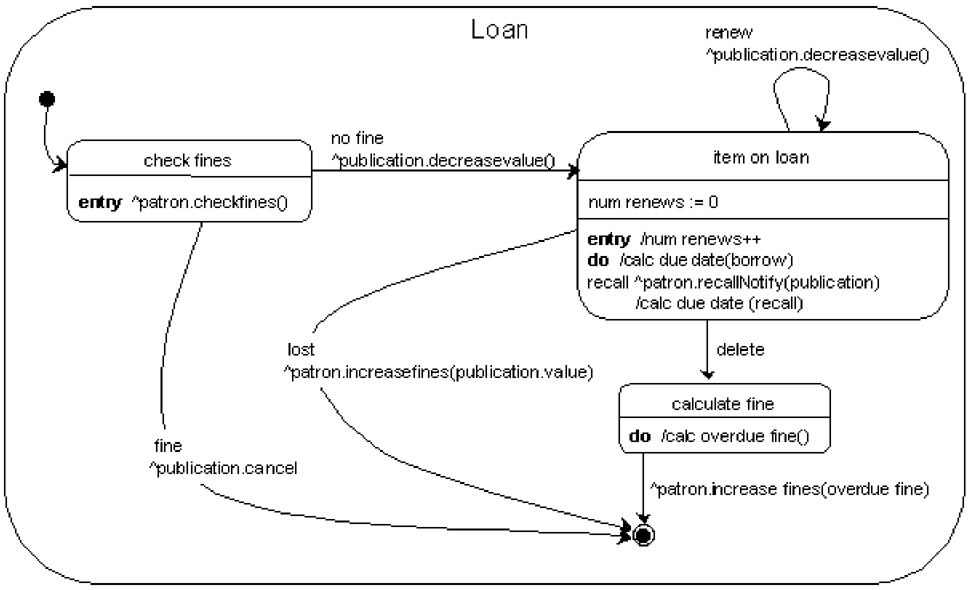

Loan
renew
^publication.decreasevalue()
check fines
entry ^patron.checkfines()
no fine
^publication.decreasevalue()
item on loan
num renews := 0
entry /num renews++
do /calc due date(borrow)
recall ^patron.recallNotify(publication)
/calc due date (recall)
lost
^patron.increasefines(publication.value)
delete
calculate fine
do /calc overdue fine()
fine
^publication.cancel
^patron.increase fines(overdue fine)

FIGURE 4.11 UML statechart diagram for Loan class.

The UML statechart diagram for the Loan association class in Figure 4.11 illustrates how states can be annotated with local variables (e.g., variable num renews), actions, and activities. Variables that are local to a state are declared and initialized in the center section of the state. The state's lower section lists actions and activities on the state's local variables as well as on the object's attributes. The distinction between actions and activities is subtle: an **action** is a computation that takes relatively no time to complete and that is uninterruptible, such as assigning an expression to a variable or sending a message. An action can be triggered by a transition entering or exiting the state, in which case it is designated by keyword **entry** or **exit** followed by arbitrarily many actions and generated events; or it can be triggered by the occurrence of an event, in which case it is designated by the event name followed by arbitrarily many actions and generated events. In our Loan statechart, variable num renews is incremented every time state item on loan is entered, that is every time the loan is renewed, and a recallNotify event is sent to the Patron whenever a recall event is received in that state. In contrast to actions, an **activity** is a more complex computation that executes over a period of time and that may be interruptible, such as executing an operation. Activities are initiated on entry to the state. When a transition, including a looping transition like the one triggered by renew, executes, the order in which actions are applied is as follows: First, the exit actions of the transition's source state are applied, followed by the transition's own actions, followed by the entry actions and activities of the new state.

The semantics of UML diagrams, and how the different diagrams fit together, are intentionally undefined, so that specifiers can ascribe semantics that best fit their problem. However, most practitioners view UML statecharts as communicating finite-state machines with first-in, first-out (FIFO) communication channels. Each object's state machine has an input queue that holds the messages sent to the object from other objects in the model or from the model's environment. Messages are stored in the input queue in the order in which they are received. In each execution step, the state machine reads the message at the head of its input queue, removing the message from the queue. The message either triggers a transition in the statechart, or it does not, in which case the message is discarded; the step *runs to completion,* meaning that the machine continues to execute enabled transitions, including transitions that wait for operations to complete, until no more transitions can execute without new input from the input queue. Thus, the machine reacts to only one message at a time.

The hardest part of constructing a state-machine model is deciding how to decompose an object's behavior into states. Some ways of thinking about states include

- Equivalence classes of possible future behavior, as defined by sequences of input events accepted by the machine: for example, every iteration of event sequence coin, push, rotated leaves the turnstile in a locked position waiting for the next visitor
- Periods of time between consecutive events, such as the time between the start and the end of an operation
- Named control points in an object's evolution, during which the object is performing some computation (e.g., state calculate fine) or waiting for some input event (e.g., state item on loan)
- Partitions of an object's behavior: for example, a book is out on loan or is in the library stacks; an item is on reserve, meaning that it can be borrowed for only short periods, or it is not on reserve

Some object properties could be modeled either as an attribute (defined in the class diagram) or as a state (defined in the object's statechart diagram), and it is not obvious which representation is best. Certainly, if the set of possible property values is large (e.g., a Patron's library fines), then it is best to model the property as an attribute. Alternatively, if the events to which the object is ready to react depend on a property (e.g., whether a book is out on loan), then it is best to model the property as a state. Otherwise, choose the representation that results in the simplest model that is easiest to understand.

Example: Petri nets

UML statechart diagrams nicely modularize a problem's dynamic behavior into the behaviors of individual class objects, with the effect that it may be easier to consider each class's behavior separately than it is to specify the whole problem in one diagram. This modularization makes it harder, though, to see how objects interact with each other. Looking at an individual statechart diagram, we can see when an object sends a message to another object. However, we have to examine the two objects' diagrams simultaneously to see that a message sent by one object can be received by the other. In fact, to be completely sure, we would have to search the possible executions (event traces) of the two machines, to confirm that whenever one object sends a message to the other, the target object is ready to receive and react to the message.

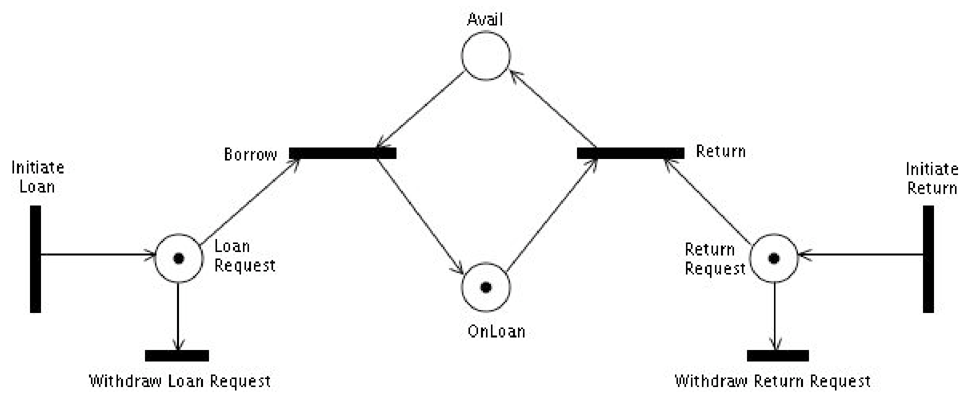


FIGURE 4.12 Petri net of book loan.

Petri nets (Peterson 1977) are a form of state-transition notation that is used to model concurrent activities and their interactions. Figure 4.12 shows a basic Petri net specifying the behavior of a book loan. The circles in the net are **places** that represent activities or conditions, and bars represent **transitions.** Directed arrows, called **arcs**, connect a transition with its *input places* and its *output places*. The places are populated with **tokens**, which act as enabling conditions for the transitions. When a transition *fires*, it removes tokens from each of its input places and inserts tokens into each its output places. Each arc can be assigned a **weight** that specifies how many tokens are removed from the arc's input place, or inserted into the arc's output place, when the transition fires. A transition is *enabled* if each of its input places contains enough tokens to contribute its arc's weight's worth of tokens, should the enabled transition actually fire. Thus, in Figure 4.12, transitions Return, Withdraw Return Request, and Withdraw Loan Request are all enabled; firing transition Return removes a token from each of the places Return Request and OnLoan, and inserts a token into Avail. The net's **marking**, which is the distribution of tokens among places, changes as transitions fire. In each execution step, the marking determines the set of enabled transitions; one enabled transition is *nondeterministically* selected to fire; the firing of this transition produces a new marking, which may enable a different set of transitions. We can model concurrent behavior by combining into a single net the activities, transitions, and tokens for several executing entities. Concurrent entities are synchronized whenever their activities, or places, act as input places to the same transition. This synchronization ensures that all of the pre-transition activities occur before the transition fires, but does not constrain the order in which these activities occur.

These features of concurrency and synchronization are especially useful for modeling events whose order of occurrence is not important. Consider the emergency room in a hospital. Before a patient can be treated, several events must occur. The triage staff must attempt to find out the name and address of the patient and to determine the patient's blood type. Someone must see if the patient is breathing, and also examine the patient for injuries. The events occur in no particular order, but all must occur before a team of doctors begins a more thorough examination. Once the treatment begins – that is, once the transition is made from a preliminary examination to a thorough one), — the doctors start new activities. The orthopedic doctors check for broken bones, while the hematologist runs blood tests and the surgeon puts stitches in a bleeding wound. The doctors' activities are independent of one another, but none can occur until the transition from the preliminary examination takes place. A state-machine model of the emergency room might specify only a single order of events, thereby excluding several acceptable behaviors, or it might specify all possible sequences of events, resulting in an overly complex model for a relatively simple problem. A Petri net model of the same emergency room nicely avoids both of these problems.

Basic Petri nets are fine for modeling how control flows through events or among concurrent entities. But if we want to model control that depends on the value of data (e.g., borrowing a particular book from a collection of books), then we need to use a high-level Petri net notation. A number of extensions to basic Petri nets have been proposed to improve the notation's expressibility, including inhibitor arcs, which enable a transition only if the input place is empty of tokens; priority among transitions; timing constraints; and structured tokens, which have values.

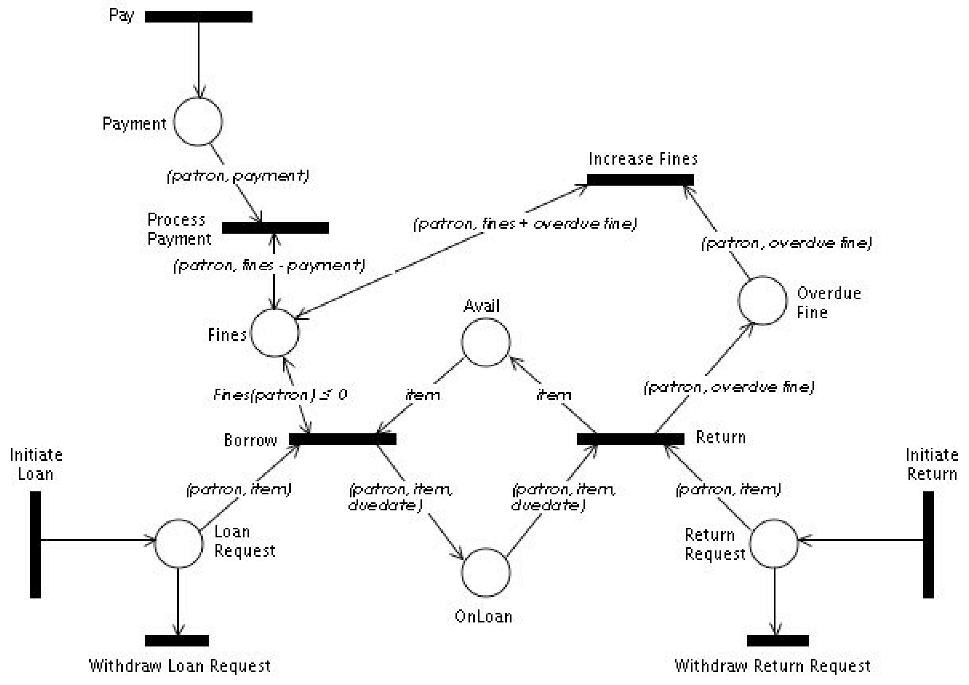


FIGURE 4.13 Petri net of the library problem.

To model our library problem, which tracks information and events for multiple patrons and publications, we need a Petri net notation that supports structured tokens and transition actions (Ghezzi et al. 1991). A transition action constrains which tokens in the input places can enable the transition and specifies the values of the output tokens. Figure 4.13 is a high-level Petri net specification for the library problem. Each place stores tokens of a different data type. Avail stores a token for every library item that is not currently out on loan. A token in Fines is an n-tuple (i.e., an ordered set of n elements, sometimes called a tuple for short) that maps a patron to the value of his or her total outstanding library fines. A token in OnLoan is another type of tuple that maps a patron and a library item to a due date. A few of the transition predicates and actions are shown in Figure 4.13, such as the action on Process Payment, where the inputs are a payment and the patron's current fines and the output is a new Fines tuple. The predicates not shown assert that if the token elements in a transition's input or output tokens have the same name, they must have the same value. So in transition Borrow, the *patron* making the Loan Request must match the *patron* with no outstanding Fines and match the *patron* who appears in the generated OnLoan tuple; at the same time, the *item* being borrowed must match an *item* in Avail. Otherwise, those tuples do not enable the transition. The net starts with an initial marking of item tuples in Avail and (patron, 0) tuples in Fines. As library users trigger input transitions Pay, Initiate Loan, and Initiate Return, new tokens are introduced to the system, which enable the library transitions, which in turn fire and update the Fines tokens and the Avail and OnLoan tokens, and so on.

Data-Flow Diagrams

The notation paradigms discussed so far promote decomposing a problem by entity (ER diagram); by scenario (event trace); and by control state (that is, equivalence classes of scenarios) (state machines). However, early requirements tend to be expressed as

- Tasks to be completed
- Functions to be computed
- Data to be analyzed, transformed, or recorded

Such requirements, when decomposed by entity, scenario, or state, devolve into collections of lower-level behaviors that are distributed among multiple entities and that must be coordinated. This modular structure makes it harder to see a model's high-level functionality. In our library example, none of the above modeling notations is effective in showing, in a single model, all of the steps, and their variants, that a patron must take to borrow a book. For this reason, notations that promote decomposition by functionality have always been popular.

A **data-flow diagram (DFD)** models functionality and the flow of data from one function to another. A bubbles represents a **process**, or function, that transforms data. An arrow represents **data flow**, where an arrow into a bubble represents an input to the bubble's function, and an arrow out of a bubble represents one of the function's outputs. Figure 4.14 shows a high-level data-flow diagram for our library problem. The problem is broken down into steps, with the results of early steps flowing into later steps. Data that persist beyond their use in a single computation (e.g., information about patrons' outstanding fines) can be saved in a **data store** – a formal repository or database of information – that is represented by two parallel bars. Data sources or sinks, represented by rectangles, are **actors**: entities that provide input data or receive the output results. A bubble can be a high-level abstraction of another data-flow diagram that shows in more detail how the abstract function is computed. A lowest-level bubble is a function whose effects, such as pre-conditions, post-conditions, exceptions, can be specified in another notation (e.g., text, mathematical functions, event traces) in a separately linked document.

One of the strengths of data-flow diagrams is that they provide an intuitive model of a proposed system's high-level functionality and of the data dependencies among the various processes. Domain experts find them easy to read and understand. However, a data-flow diagram can be aggravatingly ambiguous to a software developer who is less familiar with the problem being modeled. In particular, there are multiple ways of interpreting a DFD process that has multiple input flows (e.g., process Borrow): are all inputs needed to compute the function, is only one of the inputs needed, or is some subset of the inputs needed? Similarly, there are multiple ways of interpreting a DFD process that has multiple output flows: are all outputs generated every time the process executes, is only one of the outputs generated, or is some subset generated? It is also not obvious that two data flows with the same annotation represent the same values: are the Items Returned that flow from Return to Loan Records the same as the Items Returns that flow from Return to Process Fines? For these reasons, DFDs are best used by users who are familiar with the application domain being modeled, and as early models of a problem, when details are less important.

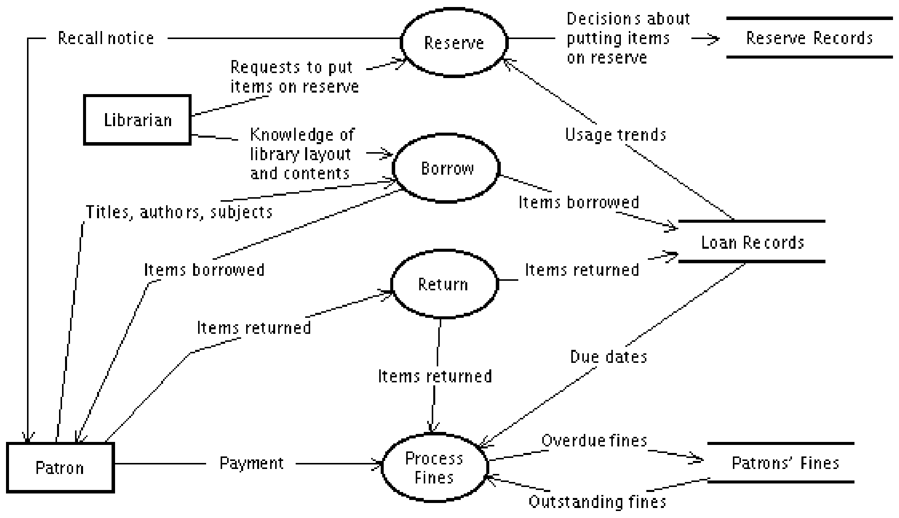


FIGURE 4.14 Data-flow diagram of the library problem.

Example: Use Cases

A UML **use-case diagram** (OMG 2003) is similar to a top-level data-flow diagram that depicts observable, user-initiated functionality in terms of interactions between the system and its environment. A large box represents the system boundary. Stick figures outside the box portray actors, both humans and systems, and each oval inside the box is a use case that represents some major required functionality and its variants. A line between an actor and a use case indicates that the actor participates in the use case. Use cases are not meant to model all tasks that the system should provide. Rather, they are used to specify user views of essential system behavior. As such, they model only system functionality that can be initiated by some actor in the environment. For example, in Figure 4.15, key library uses include borrowing a book, returning a borrowed book, and paying a library fine.

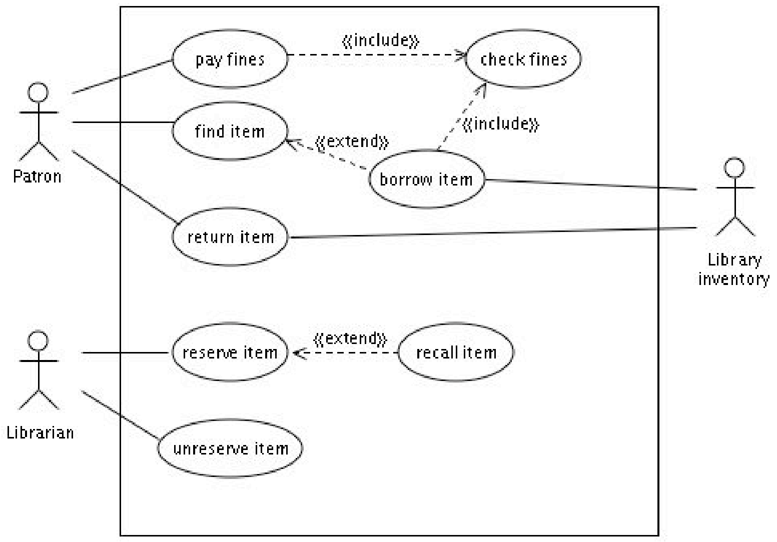


FIGURE 4.15 Library use cases.

Each use case encompasses several possible scenarios, some successful and some not, but all related to some usage of the system. External to the use-case diagram, the use cases and their variants are detailed as textual event traces. Each use case identifies pre-conditions and alternative behavior if the pre-conditions are not met, such as looking for a lost book; post-conditions, which summarize the effects of the use case; and a normal, error-free scenario comprising a sequence of steps performed by actors or by the system. A completely detailed use case specifies all possible variations of each step in the normal scenario, including both valid behaviors and errors. It also describes the possible scenarios that stem from the valid variations and from recoverable failures. If there is a sequence of steps common to several use cases, the sequence can be extracted out to form a sub case that can be called by a base use case like a procedure call. In the use-case diagram, we draw a dashed arrow from a base case to each of its sub cases and annotate these arrows with stereotype[4] ⟨⟨include⟩⟩. A use case can also be appended with an extension sub case that adds functionality to the end of the use case. In the use-case diagram, we draw a dashed arrow from the extension sub case to the base use case and annotate the arrow with stereotype ⟨⟨extend⟩⟩. Examples of stereotypes are included in Figure 4.15.

Functions and Relations

The notational paradigms discussed so far are representational and relational. They use annotated shapes, lines, and arrows to convey the entities, relationships, and characteristics involved in the problem being

[4] In UML, a **stereotype** is a meta-language facility for extending a modeling notation, allowing the user to augment one of the notation's constructs with a new ⟨⟨keyword⟩⟩.

modeled. In contrast, the remaining three notational paradigms that we discuss are more strongly grounded in mathematics, and we use them to build mathematical models of the requirements. Mathematically based specification and design techniques, called **formal methods** or **approaches,** are encouraged by many software engineers who build safety-critical systems – that is, systems whose failure can affect the health and safety of people who use them or who are nearby. For example, Defence Standard 00-56, a draft British standard for building safety-critical systems, requires that formal specification and design be used to demonstrate required functionality, reliability, and safety. Advocates argue that mathematical models are more precise and less ambiguous than other models, and that mathematical models lend themselves to more systematic and sophisticated analysis and verification. In fact, many formal specifications can be checked automatically for consistency, completeness, nondeterminism, and reachable states, as well as for type correctness. Mathematical proofs have revealed significant problems in requirements specifications, where they are more easily fixed than if revealed during testing. For example, Pfleeger and Hatton (1997) report on software developers who used formal methods to specify and evaluate the complex communications requirements of an air-traffic-control support system. Early scrutiny of the formal specification resulted in major problems that were fixed well before design began, reducing risk as well as saving development time. At the end of this chapter, we will see how formal specification might have caught problems with Ariane-5.

Some formal paradigms model requirements or software behavior as a collection of mathematical **functions** or **relations** that, when composed together, map system inputs to system outputs. Some functions specify the state of a system's execution, and other functions specify outputs. A relation is used instead of a function whenever an input value maps to more than one output value. For example, we can represent the turnstile problem using two functions: one function to keep track of the state of the turnstile, mapping from the current state and input event to the next state; and a second function to specify the turnstile's output, based on the current state and input event:

$$NextState(s,e) = \begin{cases} unlocked & s{=}locked\ AND\ e{=}coin \\ rotating & s{=}unlocked\ AND\ e{=}push \\ locked & (s{=}rotating\ AND\ e{=}rotated)\ OR\ (s{=}locked\ AND\ e{=}slug) \end{cases}$$

$$Output(s,e) = \begin{cases} buzz & s{=}locked\ AND\ e{=}slug \\ \text{<none>} & otherwise \end{cases}$$

Together, the above functions are semantically equivalent to the graphical state-machine model of the turnstile shown in Figure 4.8.

Because it maps each input to a single output, a function is by definition consistent. If the function also specifies an output for every distinct input, it is called a total function and is by definition complete. Thus, functional specifications lend themselves to systematic and straightforward tests for consistency and completeness.

Example: Decision Tables

A **decision table** (Hurley 1983) is a tabular representation of a functional specification that maps events and conditions to appropriate responses or actions. We say that the specification is *informal* because the inputs (events and conditions) and outputs (actions) may be expressed in natural language, as mathematical expressions, or both.

Put item in stacks				X				X
Put item on reserve shelf					X	X		
Send recall notice							X	
Reject event		X	X					

FIGURE 4.16 Decision table for library functions.

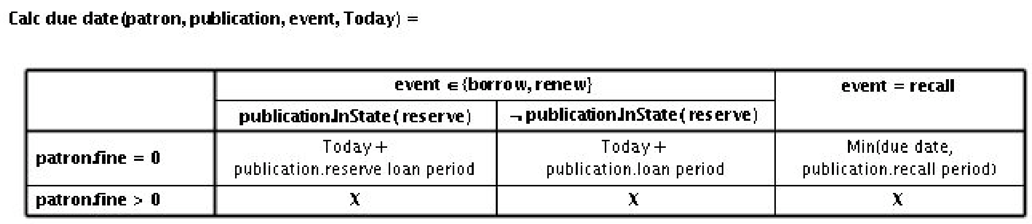
Figure 4.16 shows a decision table for the library functions borrow, return, reserve, and unreserve. All of the possible input events (i.e., function invocations), conditions, and actions are listed along the left side of the table, with the input events and conditions listed above the horizontal line and the actions listed below the line. Each column represents a rule that maps a set of conditions to its corresponding result(s). An entry of "T" in a cell means that the row's input condition is true, "F" means that the input condition is false, and a dash indicates that the value of the condition does not matter. An entry of "X" at the bottom of the table means that the row's action should be performed, whenever its corresponding input conditions hold. Thus, column 1 represents the situation where a library patron wants to borrow a book, the book is not already out on loan, and the patron has no outstanding fine; in this situation, the loan is approved and a due date is calculated. Similarly, column 7 illustrates the case where there is a request to put a book on reserve but the book is currently out on loan; in this case, the book is recalled and the due date is recalculated to reflect the recall.

This kind of representation can result in very large tables, because the number of conditions to consider is equal to the number of combinations of input conditions. That is, if there are *n* input conditions, there are 2^n possible combinations of conditions. Fortunately, many combinations map to the same set of results and can be combined into a single column. Some combinations of conditions may be infeasible (e.g., an item cannot be borrowed and returned at the same time). By examining decision tables in this way, we can reduce their size and make them easier to understand.

What else can we tell about a requirements specification that is expressed as a decision table? We can easily check whether every combination of conditions has been considered, to determine if the specification is complete. We can examine the table for consistency, by identifying multiple instances of the same input conditions and eliminating any conflicting outputs. We can also search the table for patterns to see how strongly individual input conditions correlate to individual actions. Such a search would be arduous on a specification modeled using a traditional textual notation for expressing mathematical functions.

Example: Parnas Tables

Parnas Tables (Parnas 1992) are tabular representations of mathematical functions or relations. Like decision tables, Parnas Tables use rows and columns to separate a function's definition into its different cases. Each table entry specifies either an input condition that partially identifies some case or the output value for some case. Unlike decision tables, the inputs and outputs of a Parnas Table are purely mathematical expressions.



Calc due date(patron, publication, event, Today) =

	event ∈ {borrow, renew}		event = recall
	publication.InState(reserve)	¬ publication.InState(reserve)	
patron.fine = 0	Today + publication.reserve loan period	Today + publication.loan period	Min(due date, publication.recall period)
patron.fine > 0	X	X	X

FIGURE 4.17 (Normal) Parnas Table for operation Calc due date.

To see how Parnas Tables work, consider Figure 4.17. The rows and columns define Calc due date, an operation in our library example. The information is represented as a Normal Table, which is a type of Parnas Table. The column and row headers are predicates used to specify cases, and the internal table entries store the possible function results. Thus, each internal table entry represents a distinct case in the function's definition. For example, if the event is to renew a loan (column header), *and* the publication being borrowed is on reserve (column header), *and* the patron making the request has no outstanding fines (row header), then the due date is calculated to be publication.reserve loan period days from Today. A table entry of "X" indicates that the operation is invalid under the specified conditions; in other specifications, an entry of "X" could mean that the combination of conditions is infeasible. Notice how the column and row headers are structured to cover all possible combinations of conditions that can affect the calculation of a loaned item's due date. (The symbol ¬ means "not", so ¬publication.InState(reserve) means that the publication is not on reserve).

The phrase **Parnas Tables** actually refers to a collection of table types and abbreviation strategies for organizing and simplifying functional and relational expressions. Another table type is an Inverted Table, which looks more like a conventional Decision Table: case conditions are specified as expressions in the row headers and in the table entries, and the function results are listed in the column headers, at the top or the bottom of the table. In general, the specifier's goal is to choose or create a table format that results in a simple and compact representation for the function or relation being specified. The tabular structure of these representations makes it easy for reviewers to check that a specification is complete (i.e., there are no missing cases) and consistent (i.e., there are no duplicate cases). It is easier to review each function's definition case by case, rather examining and reasoning about the whole specification at once.

A functional specification expressed using Parnas Tables is best decomposed into a single function per output variable. For every input event and for every condition on entities or other variables, each function specifies the value of its corresponding output variable. The advantage of this model structure over a state-machine model is that the definitions of each output variable is localized in a distinct table, rather than spread throughout the model as actions on state transitions.

Logic

With the exception of ER diagrams, the notations we have considered so far have been model-based and are said to be operational. An **operational** notation is a notation used to describe a problem or a proposed software solution in terms of situational behavior: how a software system should respond to different input events, how a computation should flow from one step to another, and what a system should output under various conditions. The result is a model of *case-based behavior* that is particularly useful for answering questions about what the desired response should be to a particular situation: for example, what the next state or system output should be given the current state, input event, process completion, and variable values. Such models also help the reader to visualize global behavior, in the form of paths representing allowable execution traces through the model.

Operational notations are less effective at expressing global properties or constraints. Suppose we were modeling a traffic light, and we wanted to assert that the lights controlling traffic in cross directions are never green at the same time, or that the lights in each direction are periodically green. We could build an operational model that exhibits these behaviors implicitly, in that all paths through the model satisfy these properties. However, unless the model is **closed** – meaning that the model expresses all of the desired behavior, and that any implementation that performs additional functionality is incorrect – it is ambiguous as to whether these properties are requirements to be satisfied or simply are accidental effects of the modeling decisions made.

Instead, global properties and constraints are better expressed using a descriptive notation, such as logic. A **descriptive** notation is a notation that describes a problem or a proposed solution in terms of its properties or its invariant behaviors. For example, ER diagrams are descriptive, in that they express relationship properties among entities. A **logic** consists of a language for expressing properties, plus a set of inference rules for deriving new, consequent properties from the stated properties. In mathematics, a logical expression[5], called a **formula**, evaluates to either *true* or *false*, depending on the values of the variables that appear in the formula. In contrast, when logic is used to express a property of a software problem or system, the property is an assertion about the problem or system that should be *true*. As such, a property specification represents only those values of the property's variables for which the property's expression evaluates to *true*.

There are multiple variants of logic that differ in how expressive their property notation is, or in what inference rules they provide. The logic commonly used to express properties of software requirements is **first-order logic,** comprising typed variables; constants; functions; predicates, like relational operators < and >; equality; logical connectives ∧ (and), ∨ (or), ¬ (not), ⇒ (implies), and ⇔ (logical equivalence); and quantifiers ∃ (there exists) and ∀ (for all). Consider the following variables of the turnstile problem, with their initial values:

```
num_coins : integer := 0                    /* number of coins inserted                     */
num_entries : integer := 0;                 /* number of half-rotations of turnstile        */
barrier : {locked, unlocked} := locked;     /* whether barrier is locked                    */
may_enter : boolean := false;               /* whether anyone may enter                     */
insert_coin : boolean := false;             /* event of coin being inserted                 */
push : boolean := false;                    /* turnstile is pushed sufficiently hard to rotate
                                               it one-half rotation                         */
```

The following are examples of turnstile properties over these variables, expressed in first-order logic:

$$\text{num\_coins} \geq \text{num\_entries}$$
$$(\text{num\_coins} > \text{num\_entries}) \Leftrightarrow (\text{barrier} = \text{unlocked})$$
$$(\text{barrier} = \text{locked}) \Leftrightarrow \neg\text{may\_enter}$$

Together, these formulae assert that the number of entries through the turnstile's barrier should never exceed the number of coins inserted into the turnstile, and that whenever the number of coins inserted exceeds the number of entries, the barrier is unlocked to allow another person to enter the gated area. Note that these properties say nothing about how variables change value, such as how the number of inserted coins increases. Presumably, another part of the specification describes this. The above properties simply ensure that however the variables' values change, the values always satisfy the formulae's constraints.

Temporal logic introduces additional logical connectives for constraining how variables can change value over time – more precisely, over multiple points in an execution. For example, temporal-logic connectives can express imminent changes, like variable assignments (e.g., an insert_coin event results in variable num_coins being incremented by 1), or they can express future variable values (e.g., after an insert_coin event, variable may_enter remains true until an push event). We can model this behavior in first-order logic by adding a time parameter to each of model's variables and asking about the value of a variable at a particular time. However, temporal logic, in allowing variable values to change and in introducing special temporal logic connectives, represents varying behavior more succinctly.

As with logics in general, there are many variants of temporal logic, which differ in the connectives that they introduce. The following (linear-time) connectives constrain future variable values, over a single execution trace:

[5] You can think of a logic as a function that maps expressions to a set of possible values. An *n*-valued logic maps to a set of *n* values. Binary logic maps expressions to {true, false}, but *n* can in general be larger than two. In this book, we assume that *n* is two unless otherwise stated.

$\Box$ f $\equiv$ f is *true* now and throughout the rest of the execution
$\Diamond$ f $\equiv$ f is *true* now or at some future point in the execution
$\bigcirc$ f $\equiv$ f is *true* in the next point of the execution
f W g $\equiv$ f is *true* until a point where g is *true*, but g may never be *true*

In the following, the temporal turnstile properties given above are expressed in temporal logic:

$\Box$ (insert_coin $\Rightarrow$ $\bigcirc$ (may_enter W push))
$\Box$ ($\forall$n (insert_coin $\wedge$ num_coins=n) $\Rightarrow$ $\bigcirc$ (num_coins = n+1))

Properties are often used to augment a model-based specification, either to impose constraints on the model's allowable behaviors or simply to express redundant but nonobvious global properties of the specification. In the first case, a property specifies behavior not expressed in the model, and the desired behavior is the conjunction of the model and the property. In the second case, the property does not alter the specified behavior but may aid in understanding the model by explicating otherwise implicit behavior. Redundant properties also aid in requirements verification, by providing expected properties of the model for the reviewer to check.

Example: Object Constraint Language (OCL)

The **Object Constraint Language (OCL)** is an attempt to create a constraint language that is both mathematically precise and is easy for nonmathematicians, like customers, to read, write, and understand. The language is specially designed for expressing constraints on object models (i.e., ER diagrams), and introduces language constructs for navigating from one object to another via association paths, for dealing with collections of objects, and for expressing queries on object type.

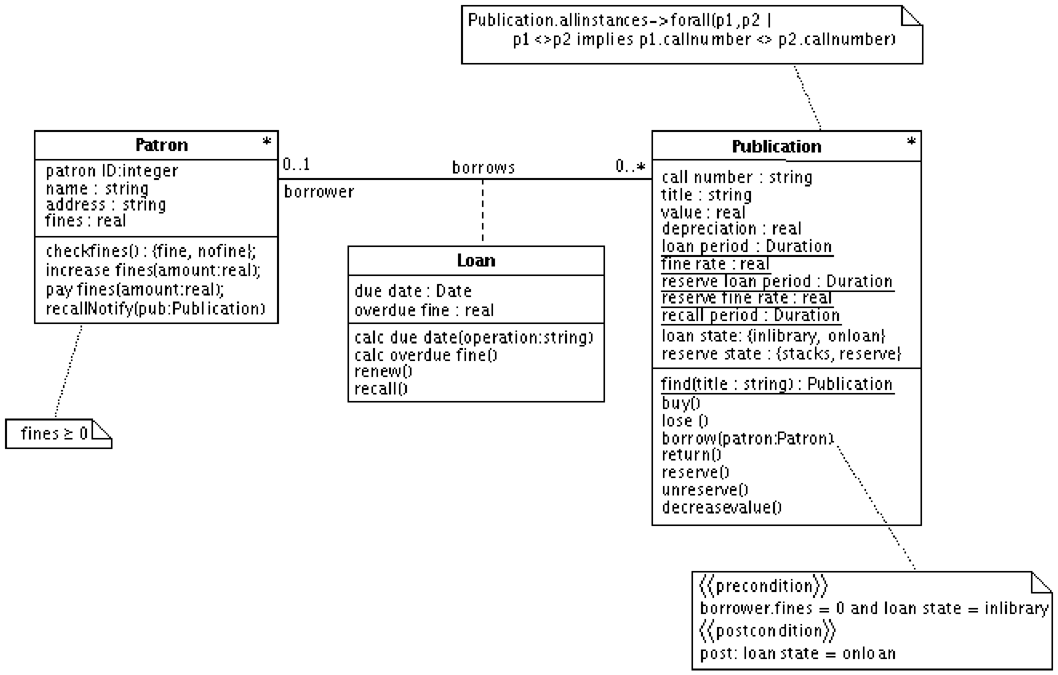


FIGURE 4.18 Library classes annotated with OCL properties.

A partial Library class model from Figure 4.5 appears in Figure 4.18, in which three classes have been detailed and annotated with OCL constraints. The leftmost constraint is an invariant on the Patron class, and specifies that no patron's fines may have a negative value (i.e., the library always makes change if a patron's payment exceeds his or her fines). The topmost constraint is an invariant on the Publication class, and specifies that call numbers are unique. This constraint introduces

- Construct *allinstances*, which returns all instances of the Publication class
- Symbol $\rightarrow$, applies the attribute or operation of its right operand to all of the objects in its left operand
- Constructs *forall, and,* and *implies*, which correspond to the first-order connectives described above.

Thus, the constraint literally says that for any two publications, p1 and p2, returned by *allinstances*, if p1 and p2 are not the same publication, then they have different call numbers. The third constraint, attached to the method borrow(), expresses the operation's pre- and post-conditions. One of the preconditions concerns an attribute in the Patron class, which is accessible via the borrow association; the patron object can be referenced either by its role name, borrower, or by the classname written in lowercase letters, if the association's far end has no role name. If the association's multiplicity were greater than 0..1, then navigating the association would return a collection of objects, and we would use $\rightarrow$ notation, rather than dot notation, to access the objects' attributes.

Although not originally designed as part of UML, OCL is now tightly coupled with UML and is part of the UML standard. OCL can augment many of UML's models. For example, it can be used to express invariants, preconditions, and post-conditions in class diagrams; invariants and transition conditions in statechart diagrams; it can also express conditions on events in message sequence charts (Warmer and Kleppe 1999). OCL annotations of UML require a relatively detailed class model, complete with attribute types, operation signatures, role names, multiplicities, and state enumerations of the class's statechart diagram. OCL expressions can appear in UML diagrams as UML notes, or they can be listed in a supporting document.

Example: Z

Z (pronounced "zed") is a formal requirements-specification language that structures set-theoretic definitions of variables into a complete abstract-data-type model of a problem, and uses logic to express the pre- and post-conditions for each operation. Z uses software-engineering abstractions to decompose a specification into manageably sized modules, called **schemas** (Spivey 1992). Separate schemas specify

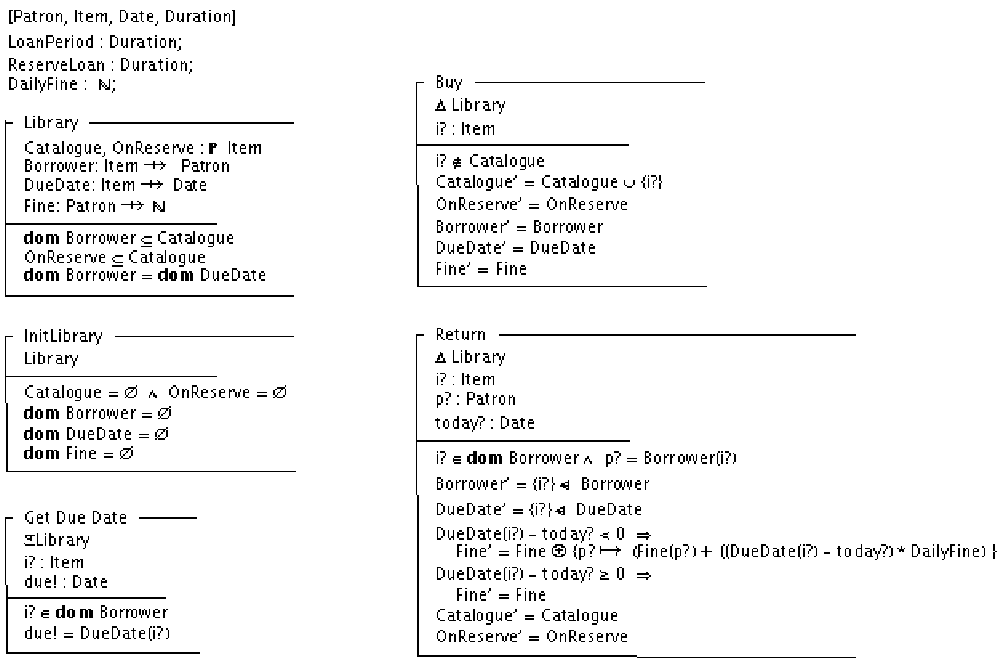
- The system state in terms of typed variables, and invariants on variables' values.
- The system's initial state (that is, initial variable values)
- Operations

Moreover, Z offers the precision of a mathematical notation and all of its benefits, such as being able to evaluate specifications using proofs or automated checks.

Figure 4.19 shows part of our library example specified in Z. Patron, Item, Date, and Duration are all basic types that correspond to their respective real-world designations. (See Sidebar 4.5 for more on designations.) The Library schema declares the problem to consist of a Catagloue and a set of items OnReserve, both declared as powersets ($\mathbb{P}$) of Items; these declarations mean that the values of Catalogue and OnReserve can change during execution to be any subset of Items. The schema also declares partial mappings ($\rightarrow\!\!\!\!+$) that record the Borrowers and DueDates for the subset of Items that are out on loan, and record Fines for the subset of Patrons who have outstanding fines. The domain (**dom**) of a partial mapping is the subset of entities currently being mapped; hence, we assert that the subset of items out on loan should be exactly the subset of items that have due dates. The InitLibrary scheme initializes all of the variables to be empty sets and functions. All of the remaining schemas correspond to library operations.

The top section of an operation schema indicates whether the operation modifies (Δ) or simply queries (Ξ) the system state, and identifies the inputs (?) and outputs (!) of the operation. The bottom section of an operation schema specifies the operation's preconditions and postconditions. In operations that modify the system state, unprimed variables represent variable values before the operation is performed, and primed variables represent values following the operation. For example, the input to operation Buy is a new library item, and the precondition specifies that the item not already be in the library Catalogue. The post-conditions update the Catalogue to include the new item, and specify that the other library variables do not change value (e.g., the updated value Fine' equals the old value Fine). Operation Return is more complicated. It takes as input the library item being returned, the

Patron who borrowed the item, and the Date of the return. The post-conditions remove the returned item from variables Borrowers and DueDates; these updates use Z symbol ⩤, "domain subtraction", to return a sub-mapping of their pre-operation values, excluding any element whose domain value is the item being returned. The next two post-conditions are updates to variable Fines, conditioned on whether the Patron incurs an overdue fine for returning the item later than the loan's due date. These updates use symbol ↦, which "maps" a domain value "to" a range value, and symbol ⊕, which "overrides" a function mapping, usually with a new "maps-to" element. Thus, if today's date is later than the returned Item's DueDate, then the Patron's fine is overridden with a new value that reflects the old fine value plus the new overdue fine. The last two post-conditions specify that the values of variable Catalogue and OnReserve do not change.



```
[Patron, Item, Date, Duration]
LoanPeriod : Duration;
ReserveLoan : Duration;
DailyFine : ℕ;

┌ Library ─────────────────────
│ Catalogue, OnReserve : ℙ Item
│ Borrower: Item ⇸ Patron
│ DueDate: Item ⇸ Date
│ Fine: Patron ⇸ ℕ
├──────────────────────
│ dom Borrower ⊆ Catalogue
│ OnReserve ⊆ Catalogue
│ dom Borrower = dom DueDate
└─────────────────────────

┌ InitLibrary ─────────────────
│ Library
├──────────────────────
│ Catalogue = ∅ ∧ OnReserve = ∅
│ dom Borrower = ∅
│ dom DueDate = ∅
│ dom Fine = ∅
└─────────────────────────

┌ Get Due Date ────────
│ ΞLibrary
│ i? : Item
│ due! : Date
├──────────────
│ i? ∈ dom Borrower
│ due! = DueDate(i?)
└──────────────────

┌ Buy ─────────────────────
│ Δ Library
│ i? : Item
├──────────────────
│ i? ∉ Catalogue
│ Catalogue' = Catalogue ∪ {i?}
│ OnReserve' = OnReserve
│ Borrower' = Borrower
│ DueDate' = DueDate
│ Fine' = Fine
└─────────────────────────

┌ Return ─────────────────────────────────
│ Δ Library
│ i? : Item
│ p? : Patron
│ today? : Date
├──────────────────
│ i? ∈ dom Borrower ∧ p? = Borrower(i?)
│ Borrower' = {i?} ⩤ Borrower
│ DueDate' = {i?} ⩤ DueDate
│ DueDate(i?) - today? < 0 ⇒
│    Fine' = Fine ⊕ {p? ↦ (Fine(p?) + ((DueDate(i?) - today?) * DailyFine) }
│ DueDate(i?) - today? ≥ 0 ⇒
│    Fine' = Fine
│ Catalogue' = Catalogue
│ OnReserve' = OnReserve
└─────────────────────────────────────
```

FIGURE 4.19 Partial Z specification of the library problem.

SIDEBAR 4.5 GROUND REQUIREMENTS IN THE REAL WORLD

Jackson's advice (Jackson 1995) to ground the requirements in the real world goes beyond the message about expressing the requirements and the specification in terms of the proposed system's environment. Jackson argues that any model of the requirements will include primitive terms that have no formal meaning (e.g., Patron, Publication, and Article in our library example), and that the only way to establish the meaning of a primitive term is to relate it to some phenomenon in the real world. He calls these descriptions **designations**, and he distinguishes *designations* from *definitions* and *assertions*. Definitions are formal meanings of terms based on the meanings of other terms used in the model (e.g., the definition of a *book out on loan*), whereas *assertions* describe constraints on terms (e.g., patrons can borrow only those items not currently out on loan). If a model is to have any meaning with respect to real-world behavior, its primitive terms must be clearly and precisely tied to the real world, and its designations "must be maintained as an essential part of the requirements documentation." (Zave and Jackson 1997)

Algebraic Specifications

With the exception of logic and OCL notations, all of the notation paradigms we have considered so far tend to result in models that suggest particular implementations. For example,

- A UML class model suggests what classes ought to appear in the final (object-oriented) implementation.
- A data-flow specification suggests how an implementation ought to be decomposed into data-transforming modules.
- A state-machine model suggests how a reactive system should be decomposed into cases.
- A Z specification suggests how complex data types can be implemented in terms of sets, sequences, or functions.

Such implementation bias in a requirements specification can lead a software designer to produce a design that adheres to the specification's *model*, subconsciously disregarding possibly better designs that would satisfy the specified *behavior*. For example, as we will see in Chapter 6, the classes in a UML class diagram may be appropriate for expressing a *problem* simply and succinctly, but the same class decomposition in a design may result in an inefficient *solution* to the problem.

A completely different way of viewing a system is in terms of what happens when combinations of operations are performed. This multi-operational view is the main idea behind **algebraic specifications:** to specify the behavior of operations by specifying the interactions between pairs of operations rather than modeling individual operations. An execution trace is the sequence of operations that have been performed since the start of execution. For example, one execution of our turnstile problem, starting with a new turnstile, is the operation sequence

```
new(). coin(). push(). rotated(). coin(). push(). rotated(). ...
```

or, in mathematical-function notation:

```
... (rotated( push( coin( rotated( push( coin( new() ) ) ) ) ) ) ) ...
```

Specification axioms specify the effects of applying pairs of operations on an arbitrary sequence of operations that have already executed (where SEQ is some prefix sequence of operations):

```
num_entries(coin(SEQ))  ≡ num_entries (SEQ)
num_entries(push(SEQ))  ≡ num_entries (SEQ)
num_entries(rotated(SEQ))  ≡  1 + num_entries (SEQ)
num_entries(new())  ≡  0
```

The first three axioms specify the behavior of operation num_entries when applied to sequences ending with operations coin, push, and rotated, respectively. For example, a rotated operation indicates that another visitor has entered the zoo, so num_entries applied to rotated(SEQ) should be one more than num_entries applied to SEQ. The fourth axiom specifies the base case of num_entries when applied to a new turnstile. Together, the four axioms indicate that operation num_entries returns, for a given sequence, the number of occurrences of the operation rotated – without saying anything about how that information may be stored or computed. Similar axioms would need to be written to specify the behaviors of other pairs of operations.

Algebraic specification notations are not popular among software developers because, for a collection of operations, it can be tricky to construct a concise set of axioms that is complete and consistent – and correct! Despite their complexity, algebraic notations have been added to several formal specification languages, to enable specifiers to define their own abstract data types for their specifications.

Example: SDL Data

SDL data definitions are used to create user-defined data types and parameterized data types in the **Specification and Description Language (SDL)** (ITU 2002). An SDL data type definition introduces the data type being specified, the signatures of all operations on that data type, and axioms that specify how pairs of operations interact. Figure 4.20 shows a partial SDL data specification for our library problem, where the library itself – the catalogue of publications, and each publication's loan and reserve status – is treated as a complex data type. NEWTYPE introduces the library data type. The LITERALS section declares any constants of the new data type; in this case, New is the value of an empty library. The OPERATORS section declares all of the library operations, including each operator's parameter types and return type. The AXIOMS section specifies the behaviors of pairs of operations.

```
NEWTYPE Library
LITERALS New;
OPERATORS
    buy: Library, Item → Library;
    lose: Library, Item → Library;
    borrow: Library, Item → Library;
    return: Library, Item → Library;
    reserve: Library, Item → Library;
    unreserve: Library, Item → Library;
    recall: Library, Item → Library;
    isInCatalogue: Library, Item → boolean;
    isOnLoan: Library, Item → boolean;
    isOnReserve: Library, Item → boolean;

 /* generators are New, buy, borrow, reserve */
```

```
AXIOMS
FOR ALL lib in Library (
  FOR ALL i, i2 in Item (
    lose(New,i) ≡ ERROR;
    lose(buy(lib, i), i2) ≡ if i=i2 then lib;
                            else buy(lose(lib, i2), i);
    lose(borrow(lib, i), i2) ≡ if i=i2 then lose(lib, i2)
                               else borrow(lose(lib, i2), i);
    lose(reserve(lib, i), i2) ≡ if i=i2 then lose(lib, i2)
                                else reserve(lose(lib, i2), i);
    return(New,i) ≡ ERROR;
    return(buy(lib, i), i2) ≡ if i=i2 then buy(lib, i);
                              else buy(return(lib, i2), i);
    return(borrow(lib, i), i2) ≡ if i=i2 then lib;
                                 else borrow(return(lib, i2), i);
    return(reserve(lib, i), i2) ≡ reserve(return(lib, i2), i);
         ...
    isInCatalogue(New,i) ≡ false;
    isInCatalogue(buy(lib, i), i2) ≡ if i=i2 then true;
                                     else isInCatalogue(lib, i2);
    isInCatalogue(borrow(lib, i), i2) ≡ isInCatalogue (lib, i2);
    isInCatalogue(reserve(lib, i), i2) ≡ isInCatalogue (lib, i2);
         ...
  )
)
ENDNEWTYPE Library;
```

FIGURE 4.20 Partial SDL data specification for the library problem.

As mentioned above, the hardest part of constructing an algebraic specification is defining a set of axioms that is complete and consistent and that reflects the desired behavior. It is especially difficult to ensure that the axioms are consistent because they are so interrelated: each axiom contributes to the specification of two operations, and each operation is specified by a collection of axioms. As such, a change to the specification necessarily implicates multiple axioms. A heuristic that helps to reduce the number of axioms, thereby reducing the risk of inconsistency, is to separate the operations into

- **Generators**, which help to build canonical representations of the defined data type
- **Manipulators**, which return values of the defined data type, but are not generators
- **Queries**, which do not return values of the defined data type

The set of generator operations is a minimal set of operations needed to construct any value of the data type. That is, every sequence of operations can be reduced to some canonical sequence of only generator operations, such that the canonical sequence represents the same data value as the original sequence. In Figure 4.20, we select New, buy, borrow, and reserve as our generator operations, because these operations can represent any state of the library, with respect to the contents of the library's catalogue, and the loan and reserve states of its publications. This decision leaves lose, return, unreserve, and renew as our manipulator operations, because they are the remaining operations that have return type Library; and leaves isInCatalogue, isOnLoan, and isOnReserve as our query operations.

The second part of the heuristic is to provide axioms that specify the effects of applying a nongenerator operation to a canonical sequence of operations. Because canonical sequences consist only of generator operations, this step means that we need to provide axioms only for pairs of operations, where each pair is a nongenerator operation that is applied to an application of a generator operation. Each axiom specifies how to reduce an operation sequence to its canonical form: applying a manipulator operation to a canonical sequence usually results in a smaller canonical sequence, because the manipulator often undoes the effects of an earlier generator operation, such as returning a borrowed book; and applying a query operation, like checking whether a book is out on loan, returns some result without modifying the already-canonical system state.

The axioms for each nongenerator operation are recursively defined:

1. There is a base case that specifies the effect of each nongenerator operation on an empty, New library. In our library specification (Figure 4.20), losing a book from an empty library is an ERROR.

2. There is a recursive case that specifies the effect of two operations on common parameters, such as buying and losing the same book. In general, such operations interact. In this case, the operations cancel each other out, and the result is the state of the library, minus the two operations. Looking at the case of losing a book that has been borrowed, we discard the borrow operation (because there is no need to keep any loan records for a lost book) and we apply the lose operation to the rest of the sequence.

3. There is a second recursive case that applies two operations to different parameters, such as buying and losing different books. Such operations do not interact, and the axiom specifies how to combine the effects of the inner operation with the result of recursively applying the outer operation to the rest of the system state. In the case of buying and losing different books, we keep the effect of buying one book, and we recursively apply the lose operation to the rest of the sequence of operations executed so far.

There is no need to specify axioms for pairs of nongenerator operations because we can use the above axioms to reduce to canonical form the application of each nongenerator operation before considering the next nongenerator operation. We could write axioms for pairs of generator operations; for example, we could specify that consecutive loans of the same book are an ERROR. However, many combinations of generator operations, such as consecutive loans of different books, will not result in reduced canonical forms. Instead, we write axioms assuming that many of the operations have preconditions that constrain when operations can be applied. For example, we assume in our library specification (Figure 4.20) that it is invalid to borrow a book that is already out on loan. Given this assumption, the effect of returning a borrowed book

```
return(borrow(SEQ,i),i)
```

is that the two operations cancel each other out and the result is equivalent to SEQ. If we do not make this assumption, then we would write the axioms so that the return operation removes *all* corresponding borrow operations:

```
return(New,i) ≡ ERROR;
return(buy(lib, i), i2) ≡ if i=i2 then buy(lib, i);
                               else buy(return(lib, i2), i);
return(borrow(lib, i), i2) ≡ if i=i2 then return(lib, i2);
                                  else borrow(return(lib, i2), i);
return(reserve(lib, i), i2) ≡ reserve(return(lib, i2), i);
```

Thus, the effect of returning a borrowed book is to discard the borrow operation and to reapply the return operation to the rest of the sequence, so that it can remove any extraneous matching borrow operations; this recursion terminates when the return operation is applied to the corresponding buy operation, which denotes the beginning of the book's existence, or to an empty library, an ERROR. Together, these axioms specify that operation return removes from the library state any trace of the item being borrowed, which is the desired behavior. A specification written in this style would nullify consecutive buying, borrowing, or reserving of the same item.

4.6 REQUIREMENTS AND SPECIFICATION LANGUAGES

At this point, you may be wondering how the software-engineering community could have developed so many types of software models with none being the preferred or ideal notation. This situation is not unlike an architect working with a collection of blueprints: each blueprint maps a particular aspect of a building's design (e.g., structural support, heating conduits, electrical circuits, water pipes) and it is the collection of plans that enables the architect to visualize and communicate the building's whole design. Each of the notational paradigms described above models problems from a different perspective: entities and relationships, traces, execution states, functions, properties, data. As such, each is the paradigm of choice for modeling a particular view of a software problem. With practice and experience, you will learn to judge which viewpoints and notations are most appropriate for understanding or communicating a given software problem.

Because each paradigm has its own strengths, a complete specification may consist of several models, each of which illustrates a different aspect of the problem. For this reason, most practical requirements and specification languages are actually combinations of several notational paradigms. By understanding the relationships between specification languages and the notational paradigms they employ, you can start to recognize the similarities among different languages and to appreciate the essential ways in which specification languages differ. At this end of this chapter, we discuss criteria for evaluating and choosing a specification language.

Unified Modeling Language (UML)

The **Unified Modeling Language (UML)** (OMG 2003) is the language best known for combining multiple notation paradigms. Altogether, the UML standard comprises eight graphical modeling notations, plus the OCL constraint language. The UML notations that are used during requirements definition and specification include

- **Use-case diagram (a high-level DFD):** A use-case diagram is used at the start of a new project, to record the essential, top-level functions that the to-be-developed product should provide. In the course of detailing the use cases' scenarios, we may identify important entities that play a role in the problem being modeled.
- **Class diagram (an ER diagram):** As mentioned previously, the class diagram is the flagship model of a UML specification, emphasizing the problem's entities and their interrelationships. The remaining UML specification models provide more detail about how the classes' objects behave and how they interact with one another. As we gain a better understanding of the problem being modeled, we detail the class diagram, with additional attributes, attribute classes, operations, and signatures. Ideally, new insight into the problem is more likely to cause changes to these details than to affect the model's entities or relationships.
- **Sequence diagram (an event trace):** Sequence diagrams are early behavioral models that depict traces of messages passed among class instances. They are best used to document important scenarios that involve multiple objects. When creating sequence diagrams, we look for common subsequences that appear in several diagrams; these subsequences may help us to identify states (e.g., the start and end points of the subsequence) in the objects' local behaviors.
- **Collaboration diagram (an event trace):** A collaboration diagram illustrates one or more event traces, overlayed on the class diagram. As such, a collaboration diagram presents the same information as a sequence diagram. The difference is that the sequence diagram emphasizes a scenario's temporal ordering of messages because it organizes messages along a timeline. On the other hand, the collaboration diagramemphasizes the classes' relationships, and treats the messages as elaborations of those relationships in the way that it represents messages as arrows between classes in the class diagram.
- **Statechart diagram (a state-machine model):** A UML statechart diagram specifies how each instance of one class in the specification's class diagram behaves. Before writing a statechart for a class (more specifically, for a representative object of that class), we should identify the states in the object's lifecyle, the events that this object sends to and receives from other objects, the order in which these states and events occur, and the operations that the object invokes. Because such information is fairly detailed, a statechart diagram should not be attempted until late in the requirements phase, when the problem's details are better understood.
- **OCL properties (logic):** OCL expressions are properties about a model's elements (e.g., objects, attributes, events, states, messages). OCL properties can be used in any of the above models, to explicate the model's implicit behavior or to impose constraints on the model's specified behavior.

Most of these notations were discussed in the previous section, as examples of different notation paradigms. In Chapter 6, we will see more details about how UML works, by applying it to a real-world problem for both specification and design.

Specification and Description Language (SDL)

The **Specification and Description Language (SDL)** (ITU 2002) is a language standardized by the International Telecommunications Union for specifying precisely the behavior of real-time, concurrent, distributed processes that communicate with each other via unbounded message queues. SDL comprises three graphical diagrams, plus algebraic specifications for defining complex data types:

- **SDL system diagram (a DFD):** An SDL system diagram, shown in Figure 4.21(a), depicts the top-level blocks of the specification and the communication channels that connect the blocks. The channels are directional and are labeled with the types of signals that can flow in each direction. Message passing via **channels** is *asynchronous*, meaning that we cannot make any assumptions about when sent messages will be received; of course, messages sent along the *same* channel will be received in the order in which they were sent.
- **SDL block diagram (a DFD):** Each SDL block may model a lower-level collection of blocks and the message-delaying channels that interconnect them. Alternatively, it can model a collection of lowest-level processes that communicate via signal routes, shown in Figure 4.21(b). **Signal routes** pass messages *synchronously*, so messages sent between processes in the same block are received instantaneously. In fact, this difference in communication mechanisms is a factor when deciding how to decompose behavior: processes that need to synchronize with one another and that are highly coupled should reside in the same block.
- **SDL process diagram (a state-machine model):** An SDL process, shown in Figure 4.21(c), is a state machine, whose transitions are sequences of language constructs (input, decisions, tasks, outputs) that start and end at state constructs. Each process has a single input queue that holds all of the unread messages that the process has received via all of its signal routes. In each execution step, the process removes a signal from the head of its input queue; compares the signal to the input constructs that follow the process's current state and that specify the input events to which the process is ready to react; and, if the signal matches one of the state's inputs, executes all of the constructs that follow the matching input, until the execution reaches the next state construct.
- **SDL data type (algebraic specification):** An SDL process may declare local variables, and SDL data type definitions are used to declare complex, user-defined variable types.

In addition, an SDL specification is often accompanied by a set of Message Sequence Charts (MSC) (ITU 1996), each of which illustrates a single execution of the specification in terms of the messages passed among the specification's processes.

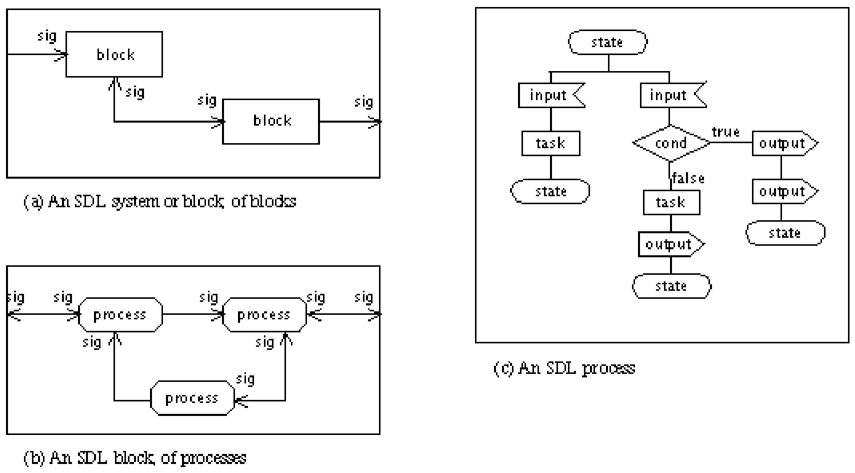


FIGURE 4.21 SDL graphical notations.

Software Cost Reduction (SCR)

Software Cost Reduction (SCR) (Heitmeyer 2002) is a collection of techniques that were designed to encourage software developers to employ good software engineering design principles. An SCR specification models software requirements as a mathematical function, REQ, that maps **monitored variables**, which are environmental variables that are *sensed* by the system, to **controlled variables,** which are environmental variables that are *set* by the system. The function REQ is decomposed into a collection of tabular functions, similar to Parnas Tables. Each of these functions is responsible for setting the value of one controlled variable or the value of a **term,** which is a macro-like variable that is referred to in other functions' definitions.

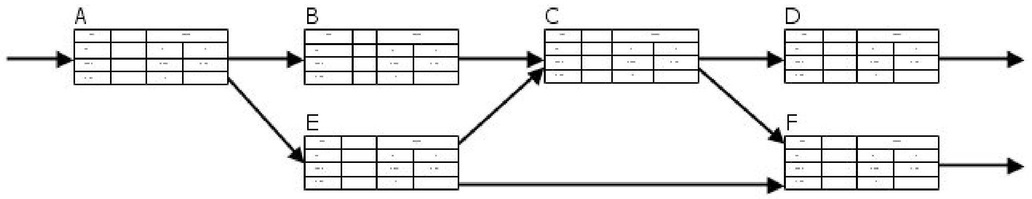


FIGURE 4.22 SCR specification as a network of tabular functions.

REQ is the result of composing these tabular functions into a network (a DFD, as shown in Figure 4.22), whose edges reflect the data dependencies among the functions. Every execution step starts with a change in the value of one monitored variable. This change is then propagated through the network, in a single, synchronized step. The specification's functions are applied in a topological sort that adheres to the functions' data dependencies: any function that refers to updated values of variables must execute *after* the functions that update those values. Thus, an execution step resembles a wave of variable updates flowing through the network, starting with newly sensed monitored-variable values, followed by updates to term variables, followed by updates to controlled-variable values.

Other Features of Requirements Notations

There are many other requirements-modeling techniques. Some techniques include facilities for associating the degree of uncertainty or risk with each requirement. Other techniques have facilities for tracing requirements to other system documents, such as design or code, or to other systems, such as when requirements are reused. Most specification techniques have been automated to some degree, making it easy to draw diagrams, collect terms and designations into a data dictionary, and check for obvious inconsistencies. As tools continue to be developed to aid software engineering activities, documenting and tracking requirements will be made easier. However, the most difficult part of requirements analysis — understanding our customers' needs — is still a human endeavor.

4.7 Prototyping Requirements

When trying to determine requirements, we may find that our customers are uncertain of exactly what they want or need. Elicitation may yield only a "wish list" of what the customers would like to see, with few details or without being clear as to whether the list is complete. Beware! These same customers, who are indecisive in their requirements, have no trouble distinguishing between a delivered system that meets their needs and one that does not – known also as "I'll know it when I see it" customers (Boehm 2000). In fact, most people find it easier to critique, in detail, an existing product than to imagine, in detail, a new product. As such, one way that we can elicit details is to build a prototype of the proposed system and to solicit feedback from potential users about what aspects they would like to see improved, which features are not so useful, or what functionality is missing. Building a prototype can also help us determine whether the customer's problem has a feasible solution, or assist us in exploring options for optimizing quality requirements.

To see how prototyping works, suppose we are building a tool to track how much a user exercises each day. Our customers are exercise physiologists and trainers, and their clients will be the users. The tool will help the physiologists and trainers to work with their clients and to track their clients' training progress. The tool's user interface is important, because the users may not be familiar with computers. For example, in entering information about their exercise routines, the users will need to enter the date for each routine. The trainers are not sure what this interface should look like, so we build a quick prototype to demonstrate the possibilities. Figure 4.23 shows a first prototype, in which the user must type the day, month, and year. A more interesting and sophisticated interface involves a calendar (see Figure 4.24), where the user uses a mouse to select the month and year, the chart for that month is displayed, and the user selects the appropriate day in the chart. A third alternative is depicted in Figure 4.25, in which, instead of a calendar, the user is presented with three slider bars. As the user uses the mouse to slide each bar left or right, the box at the bottom of the screen changes to show the selected day, month, and year. This third interface may provide the fastest selection, even though it may be very different from what the users are accustomed to seeing. In this example, prototyping helps us to select the right "look and feel" for the user's interaction with the proposed system. The prototype interfaces would be difficult to describe in words or symbols, and they demonstrate how some types of requirements are better represented as pictures or prototypes.

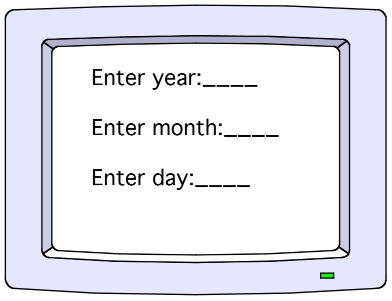


FIGURE 4.23 Keyboard-entry prototype.

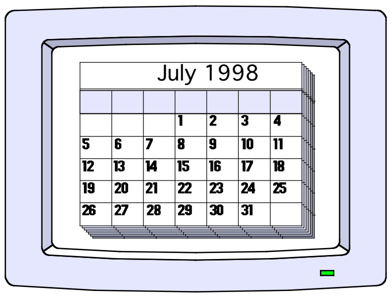


FIGURE 4.24 Calendar-based prototype. [Note to pubs: Please change artwork at top of calendar so that it reads "July 2006" instead of "Julv 1998"]

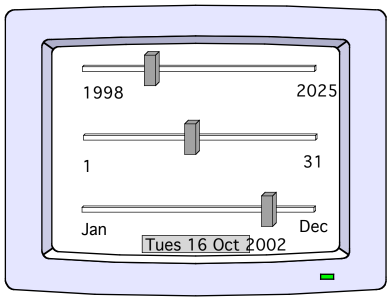


FIGURE 4.25 Slide-bar–based prototype. [Note to pubs: Please change artwork in two places, so that it reads "2000" instead of "1998" at the upper left, and so that it reads "Tues 16 Oct 2006" at the bottom, instead of "Tues 16 Oct 2002"]

There are two approaches to prototyping: throw-away and evolutionary. A **throw-away prototype** is software that is developed to learn more about a problem or about a proposed solution, and that is never intended to be part of the delivered software. This approach allows us to write "quick-and-dirty" software that is poorly structured, inefficient, with no error checking – that, in fact, may be a façade that does not implement any of the desired functionality – but that gets quickly to the heart of questions we have about the problem or about a proposed solution. Once our questions are answered, we throw away the prototype software and start engineering the software that will be delivered. In contrast, an **evolutionary prototype** is software that is developed not only to help us answer questions but also to be incorporated into the final product. As such, we have to be much more careful in its development, because this software has to eventually exhibit the quality requirements (e.g., response rate, modularity) of the final product, and these qualities cannot be retrofitted.

Both techniques are sometimes called **rapid prototyping**, because they involve building software in order to answer questions about the *requirements*. The term "rapid" distinguishes software prototyping from that in other engineering disciplines, in which a prototype is typically a complete solution, like a prototype car or plane that is built manually according to an already approved design. The purpose of such a prototype is test the design and product before automating or optimizing the manufacturing step for mass production. In rapid prototyping, a prototype is a partial solution that is built to help us understand the requirements or to evaluate design alternatives.

Questions about the requirements can be explored via either modeling or prototyping. Whether one approach is better than the other depends on what our questions are, how well they can be expressed in models or in software, and how quickly the models or prototype software can be built. As we saw above, questions about user interfaces may be easier to answer using prototypes. A prototype that implements a number of proposed features would more effectively help users to prioritize these features, and possibly to identify some features that are unnecessary. On the other hand, questions about constraints on the order in which events should occur, or about the synchronization of activities, can be answered more quickly using models. In the end, we need to produce final requirements documentation for the testing and maintenance teams, and possibly for regulatory bodies, as well as final software to be delivered. So, whether it is better to model or to prototype depends on whether it is faster and easier to model, and to develop the software from the refined models, or faster and easier to prototype, and to develop documentation from the refined prototype.

4.8 Requirements Documentation

No matter what method we choose for defining requirements, we must keep a set of documents recording the result. We and our customers will refer to these documents throughout development and maintenance. Therefore, the requirements must be documented so that they are useful not only to the customers but also to the technical staff on our development team. For example, the requirements must be organized in such a way that they can be tracked throughout the system's development. Clear and precise illustrations and diagrams accompanying the documentation should be consistent with the text. Also, the level at which the requirements are written is important, as explained in Sidebar 4.6.

SIDEBAR 4.6 LEVEL OF SPECIFICATION

In 1995, the Australian Defence and Technology Organisation reported the results of a survey of problems with requirements specifications for Navy software (Gabb and Henderson 1995). One of the problems it highlighted was the uneven level of specifications. That is, some requirements had been specified at too high a level and others were too detailed. The unevenness was compounded by several situations:

- Requirements analysts used different writing styles, particularly in documenting different system areas.
- The difference in experience among analysts led to different levels of detail in the requirements.
- In attempting to reuse requirements from previous systems, analysts used different formats and writing styles.
- Requirements were often overspecified in that analysts identified particular types of computers and programming languages, assumed a particular solution, or mandated inappropriate processes and protocols. Analysts sometimes mixed requirements with partial solutions, leading to "serious problems in designing a cost-effective solution."
- Requirements were sometimes underspecified, especially when describing the operating environment, maintenance, simulation for training, administrative computing, and fault tolerance.

Most of those surveyed agreed that there is no universally correct level of specification. Customers with extensive experience prefer high-level specifications, and those with less experience like more detail. The survey respondents made several recommendations, including:

- Write each clause so that it contains only one requirement.
- Avoid having one requirement refer to another requirement.
- Collect similar requirements together

The requirements definition is a record of the requirements expressed in the customer's terms. Working with the customer, we document what the customer can expect of the delivered system:

1. First, we outline the general purpose and scope of the system, including relevant benefits, objectives, and goals. References to other related systems are included, and we list any terms, designations, and abbreviations that may be useful.
2. Next, we describe the background and the rationale behind the proposal for a new system. For example, if the system is to replace an existing approach, we explain why the existing approach is unsatisfactory. Current methods and procedures are outlined in enough detail so that we can separate those elements with which the customer is happy from those that are disappointing.
3. Once we record this overview of the problem, we describe the essential characteristics of an acceptable solution. This record includes brief descriptions of the product's core functionality, at the level of Use Cases. It also includes quality requirements, such as timing, accuracy, and responses to failures. Ideally, we would prioritize these requirements and identify those that can be put off to later versions of the system.
4. As part of the problem's context, we describe the environment in which the system will operate. We list any known hardware and software components with which the proposed system will have to interact. To help ensure that the user interface is appropriate, we sketch the general backgrounds and capabilities of the intended users, such as their educational background, experience, and technical expertise. For example, we would devise different user interfaces for knowledgeable users than we would for first-time users. In addition, we list any known constraint on the requirements or the design, such as applicable laws, hardware limitations, audit checks, regulatory policies, and so on.
5. If the customer has a proposal for solving the problem, we outline a description of his proposal. Remember, though, that the purpose of the requirements documents is to discuss the problem, not the solution. We need to evaluate the proposed solution carefully, to determine if it is a design constraint to be satisfied or if it is an overspecification that could exclude better solutions. In the end, if the customer places any constraints on the development or if there are any special assumptions to be made, they should be incorporated into the requirements definition.

SIDEBAR 4.7 HIDDEN ASSUMPTIONS

Zave and Jackson (1997) have looked carefully at problems in software requirements and specification, including undocumented assumptions about how the real world behaves.

There are actually two types of environmental behavior of interest: desired behavior to be realized by the proposed system (i.e., the requirements) and existing behavior that is unchanged by the proposed system. The latter type of behavior is often called **assumptions** or **domain knowledge.** Most requirements writers consider assumptions to be simply the conditions under which the system is guaranteed to operate correctly. While necessary, these conditions are not the only assumptions. We also make assumptions about how the environment will behave in response to the system's outputs.

Consider a railroad-crossing gate at the intersection of a road and a set of railroad tracks. Our requirement is that trains and cars do not collide in the intersection. However, the trains and cars are outside the control of our system; all our system can do is lower the crossing gate upon the arrival of a train and lift the gate after the train passes. The only way our crossing gate will prevent collisions is if trains and cars follow certain rules. For one thing, we have to assume that the trains travel at some maximum speed, so that we know how early to lower the crossing gate to ensure that the gate is down well before a sensed train reaches the intersection. But we also have to make assumptions about how car drivers will react to the crossing gate being lowered: we have to assume that cars will not stay in or enter the intersection when the gate is down.

6. Finally, we list any assumptions we make about how the environment behaves. In particular, we describe any environmental conditions that would cause the proposed system to fail, and any changes to the environment that would cause us to change our requirements. Sidebar 4.6 explains in more detail why it is important to document assumptions. The assumptions should be documented separately from the requirements, so that developers know which behaviors they are responsible for implementing.

Requirements Specification

The requirements specification covers exactly the same ground as the requirements definition, but from the perspective of the developers. Where the requirements definition is written in terms of the customer's vocabulary, referring to objects, states, events, and activities in the customer's world, the requirements specification is written in terms of the system's interface. We accomplish this by rewriting the requirements so that they refer only to those real-world objects (states, events, actions) that are sensed or actuated by the proposed system:

1. In documenting the system's interface, we describe all inputs and outputs in detail, including the sources of inputs, the destinations of outputs, the value ranges and data formats of input and output data, protocols governing the order in which certain inputs and outputs must be exchanged, window formats and organization, and any timing constraints. Note that the user interface is rarely the only system interface; the system may interact with other software components (e.g., a database), special-purpose hardware, the Internet, and so on.
2. Next, we restate the required functionality in terms of the interfaces' inputs and outputs. We may use a functional notation or data-flow diagrams to map inputs to outputs, or use logic to document functions' pre-conditions and post-conditions. We may use state machines or event traces to illustrate exact sequences of operations or exact orderings of inputs and outputs. We may use an entity-relationship diagram to collect related activities and operations into classes. In the end, the specification should be complete, meaning that it should specify an output for any feasible sequence of inputs. Thus, we include validity checks on inputs and system responses to exceptional situations, such as violated pre-conditions.
3. Finally, we devise fit criteria for each of the customer's quality requirements, so that we can conclusively demonstrate whether our system meets these quality requirements.

The result is a description of what the developers are supposed to produce, written in sufficient detail to distinguish between acceptable and unacceptable solutions, but without saying how the proposed system should be designed or implemented:

1. Introduction to the Document
    - 1.1 Purpose of the Product
    - 1.2 Scope of the Product
    - 1.3 Acronyms, Abbreviations, Definitions
    - 1.4 References
    - 1.5 Outline of the rest of the SRS
2. General Description of Product
    - 2.1 Context of Product
    - 2.2 Product Functions
    - 2.3 User Characteristics
    - 2.4 Constraints
    - 2.5 Assumptions and Dependencies
3. Specific Requirements
    - 3.1 External Interface Requirements
        - 3.1.1 User Interfaces
        - 3.1.2 Hardware Interfaces
        - 3.1.3 Software Interfaces
        - 3.1.4 Communications Interfaces
    - 3.2 Functional Requirements
        - 3.2.1 Class 1
        - 3.2.2 Class 2
        - …
    - 3.3 Performance Requirements
    - 3.4 Design Constraints
    - 3.5 Quality Requirements
    - 3.6 Other Requirements
4. Appendices

FIGURE 4.26 IEEE standard for Software Requirements Specification organized by object (IEEE 1998).

Several organizations, such as the IEEE and the U.S. Department of Defense, have standards for the content and format of the requirements documents. For example, Figure 4.26 shows a template based on IEEE's recommendations for organizing a software requirements specification by classes or objects. The IEEE standard provides similar templates for organizing the requirements specification by mode of operation, function, feature, category of user, and so on. You may want to consult these standards in preparing documents for your own projects.

Process Management and Requirements Traceability

There must be a direct correspondence between the requirements in the definition document and those in the specification document. It is here that the process management methods used throughout the life cycle begin. **Process management** is a set of procedures that track

- The requirements that define what the system should do
- The design modules that are generated from the requirements
- The program code that implements the design
- The tests that verify the functionality of the system
- The documents that describe the system

In a sense, process management provides the threads that tie the system parts together, integrating documents and artifacts that have been developed separately. These threads allow us to coordinate the development activities, as shown by the horizontal "threads" among entities in Figure 4.27. In particular, during requirements activities, we are concerned about establishing a correspondence between elements of the requirements definition and those of the requirements specification, so that the customer's view is tied to the developer's view in an organized, traceable way. If we do not define these links, we have no way of designing test cases to determine if the code meets the requirements. In later chapters, we will see how process management also allows us to determine the impact of changes, as well as to control the effects of parallel development.

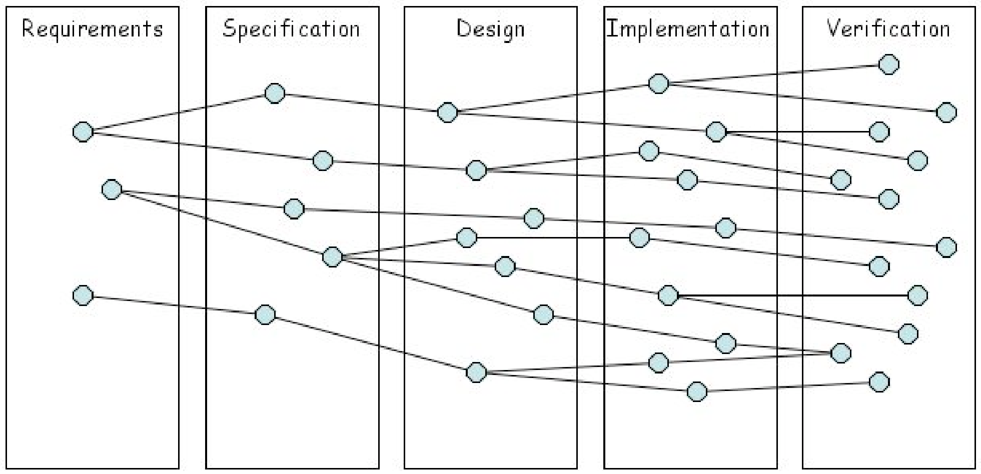


FIGURE 4.27 Links between software-development entities.

To facilitate this correspondence, we establish a numbering scheme or data file for convenient tracking of requirements from one document to another. Often, the process management team sets up or extends this numbering scheme, to tie requirements to other components and artifacts of the system. Numbering the requirements allows us to cross-reference them with the data dictionary and other supporting documents. If any changes are made to the requirements during the remaining phases of development, the changes can be tracked from the requirements document through the design process and all the way to the test procedures. Ideally, then, it should be possible to trace any feature or function of the system to its causal requirement, and vice versa.

4.9 Validation and Verification

Remember that the requirements documents serve both as a contract between us and the customer, detailing what we are to deliver, and as guidelines for the designers, detailing what they are to build. Thus, before the requirements can be turned over to the designers, we and our customers must be absolutely sure that each knows the other's intent, and that our intents are captured in the requirements documents. To establish this certainty, we *validate* the requirements and *verify* the specification.

We have been using the terms "verify" and "validate" throughout this chapter without formally defining them. In **requirements validation**, we check that our requirements definition accurately reflects the customer's – actually, all of the stakeholders' – needs. Validation is tricky because there are only a few documents that we can use as the basis for arguing that the requirements definitions are correct. In **verification**, we check that one document or artifact conforms to another. Thus, we verify that our code conforms to our design, and that our design conforms to our requirements specification; at the requirements level, we verify that our requirements specification conforms to the requirements definition. To summarize, verification ensures that we *build the system right,* whereas validation ensures that we *build the right system!*

Requirements Validation

Our criteria for validating the requirements are the characteristics that we listed in Section 4.4:

- Correct
- Consistent
- Unambiguous
- Complete
- Relevant

TABLE 4.3 Validation and Verification Techniques

Validation	Walkthroughs Reading Interviews Reviews Checklists Models to check functions and relationships Scenarios Prototypes Simulation Formal inspections
Verification	Cross-referencing Simulation Consistency checks Completeness checks Checks for unreachable states or transitions Model checking Mathematical proofs

- Testable
- Traceable

Depending on the definition techniques that we use, some of the above checks (e.g., that the requirements are consistent or are traceable) may be automated. Also, common errors can be recorded in a **checklist**, which reviewers can use to guide their search for errors. Lutz (1993) reports on the success of using checklists in validating requirements at NASA's Jet Propulsion Laboratory. However, most validation checks (e.g., that a requirement is correct, relevant, or unambiguous; or that the requirements are complete) are subjective exercises — in that they involve comparing the requirements definition against the stakeholders' mental model of what they expect the system to do. For these validation checks, our only recourse is to rely on the stakeholders' assessment of our documents.

Table 4.3 lists some of the techniques that can be used to validate the requirements. Validation can be as simple as reading the document and reporting errors. We can ask the validation team to sign off on the document, thereby declaring that they have reviewed the document and that they approve it. By signing off, the stakeholders accept partial responsibility for errors that are subsequently found in the document. Alternatively, we can hold a **walkthrough**, in which one of the document's authors presents the requirements to the rest of the stakeholders, and asks for feedback. Walkthroughs work best when there are a large number of varied stakeholders, and it is unrealistic to ask them all to examine the document in detail. At the other extreme, validation can be as structured as a **formal inspection**, in which reviewers take on specific roles (e.g., presenter, moderator) and follow prescribed rules (e.g., rules on how to examine the requirements, when to meet, when to take breaks, whether to schedule a follow-up inspection).

More often, the requirements are validated in a requirements **review.** In a review, representatives from our staff and the customer's staff examine the requirements document individually and then meet to discuss identified problems. The customer's representatives include those who will be operating the system, those who will prepare the system's inputs, and those who will use the system's outputs; managers of these employees may also attend. We provide members of the design team, the test team, and the process team. By meeting as a group, we can do more than check that the requirements definition satisfies the validation criteria:

1. We review the stated goals and objectives of the system.
2. We compare the requirements with the goals and objectives, to make certain that all requirements are necessary.
3. We review the environment in which the system is to operate, examining the interfaces between our proposed system and all other systems and checking that their descriptions are complete and correct.

4. The customer's representatives review the information flow and the proposed functions, to confirm that they accurately reflect the customer's needs and intentions. Our representatives review the proposed functions and constraints, to confirm that they are realistic and within our development abilities. All requirements are checked again for omissions, incompleteness, and inconsistency.
5. If any risk is involved in the development or in the actual functioning of the system, we can assess and document this risk, discuss and compare alternatives, and come to some agreement on the approaches to be used.
6. We can talk about testing the system: how the requirements will be revalidated, as the requirements grow and change; who will provide test data to the test team; which requirements will be tested in which phases, if the system is to be developed in phases.

SIDEBAR 4.8 NUMBER OF REQUIREMENTS FAULTS

How many development problems are created during the process of capturing requirements? There are varying claims. Boehm and Papaccio (1988), in a paper analyzing software at IBM and TRW, found that most errors are made during design, and there are usually three design faults for every two coding faults. They point out that the high number of faults attributed to the design stage could derive from requirements errors. In his book on software engineering economics, Boehm (1981) cites studies by Jones and Thayer and others that attribute

- 35% of the faults to design activities for projects of 30,000-35,000 delivered source instructions
- 10% of the faults to requirements activities and 55% of the faults to design activities for projects of 40,000-80,000 delivered source instructions
- 8% to 10% of the faults to requirements activities and 40% to 55% of the faults to design activities for projects of 65,000-85,000 delivered source instructions

Basili and Perricone (1984), in an empirical investigation of software errors, report that 48% of the faults observed in a medium-scale software project were "attributed to incorrect or misinterpreted functional specifications or requirements."

Beizer (1990) attributes 8.12% of the faults in his samples to problems in functional requirements. He includes in his count problems such as incorrect requirements; illogical or unreasonable requirements; ambiguous, incomplete, or overspecified requirements; unverifiable or untestable requirements; poorly presented requirements; and changed requirements. However, Beizer's taxonomy includes no design activities. He says, "Requirements, especially expressed in a specification (or often, as not expressed because there is no specification) are a major source of expensive bugs. The range is from a few percent to more than 50%, depending on application and environment. What hurts most about these bugs is that they're the earliest to invade the system and the last to leave. It's not unusual for a faulty requirement to get through all development testing, beta testing, and initial field use, only to be caught after hundreds of sites have been installed."

Other summary statistics abound. For example, Perry and Stieg (1993) conclude that 79.6% of interface faults and 20.4% of the implementation faults are due to incomplete or omitted requirements. Similarly, Computer Weekly Report (1994) discussed a study showing that 44.1% of all system faults occurred in the specification stage. Lutz (1993b) analyzed safety-related errors in two NASA spacecraft software systems, and found that "the primary cause of safety-related interface faults is misunderstood hardware interface specifications" (48% to 67% of such faults), and the "primary cause of safety-related functional faults is errors in recognizing (understanding) the requirements" (62% to 79% of such faults). What is the right number for your development environment? Only careful record keeping will tell you. These records can be used as a basis for measuring improvement as you institute new practices and use new tools.

Whenever a problem is identified, the problem is documented, its cause is determined, and the requirements analysts are charged with the task of fixing the problem. For example, validation may reveal that there is a great misunderstanding about the way in which a certain function will produce results. The customers may require data to be reported in miles, whereas the users want the data in kilometers. The customers may set a reliability or availability goal that developers deem impossible to meet. These conflicts need to be resolved before design can begin. To resolve a conflict, the developers may need to construct simulations or prototypes to explore feasibility constraints and then work with the customer to agree on an acceptable requirement. Sidebar 4.8 discusses the nature and number of requirements-related problems you are likely to find.

Our choice of validation technique depends on the experience and preferences of the stakeholders and on the technique's appropriateness for the notations used in the requirements definition. Some notations have tool support for checking consistency and completeness. There are also tools that can help with the review process and with tracking problems and their resolutions. For example, some tools work with you and the customer to reduce the amount of uncertainty in the requirements. This book's Web page points to requirements-related tools.

Verification

In verification, we want to check that our requirements specification document corresponds to our requirements definition document. This verification makes sure that if we implement a system that meets the specification, then that system will satisfy the customer's requirements. Most often, this is simply a check of traceability, where we ensure that each requirement in the definition document is traceable to the specification.

However, for critical systems, we may want to do more, and actually demonstrate that the specification fulfills the requirements. This is a more substantial effort, in which we prove that the specification realizes every function, event, activity, and constraint in the requirements. The specification by itself is rarely enough to make this kind of argument, because the specification is written in terms of actions performed at the system's interface, such as force applied to an unlocked turnstile, and we may want to prove something about the environment away from the interface, such as about the number of entries into the zoo. To bridge this gap, we need to make use of our assumptions about how the environment behaves – assumptions about what inputs the system will receive, or about how the environment will react to outputs (e.g., that if an unlocked turnstile is pushed with sufficient force, it will rotate one-half a turn, nudging the pusher into the zoo). Mathematically, the specification (S) plus our environmental assumptions (A) must be sufficient to prove that the requirements (R) hold:

$$S, A \vdash R$$

For example, to show that a thermostat and furnace will control air temperature, we have to assume that air temperature changes continuously rather than abruptly, although the sensors may detect discrete value changes, and that an operating furnace will raise the air temperature. These assumptions may seem obvious, but if a building is sufficiently porous and the outside temperature is sufficiently cold, then our second assumption will not hold. In such a case, it would be prudent to set some boundaries on the requirement: as long as the outside temperature is above –100C, the thermostat and furnace will control the air temperature.

This use of environmental assumptions gets at the heart of why their documentation is so important: we rely on the environment to help us satisfy the customer's requirements, and if our assumptions about how the environment behaves are wrong, then our system may not work as the customer expects. If we cannot prove that our specification and our assumptions fulfill the customer's requirements, then we need either to change our specification, strengthen our assumptions about the environment, or weaken the requirements we are trying to achieve. Sidebar 4.9 discusses some techniques for automating these proofs.

When requirements validation and verification is complete, we and our customers should feel comfortable about the requirement specification. Understanding what the customer wants, we can proceed with the system design. Meanwhile, the customer has in hand a document describing exactly what the delivered system should do.

SIDEBAR 4.9 COMPUTER-AIDED VERIFICATION

Model checking is an exhaustive search of a specification's execution space, to determine whether some temporal-logic property holds of the executions. The model checker computes and searches the specification's execution space, sometimes computing the execution space symbolically and sometimes computing it on-the-fly during the search. Thus, the verification, while completely automated, consumes significant computing resources. Sreemani and Atlee (1996) used the SMV model checker to verify five properties of an SCR specification of the A-7 naval aircraft. Their SMV model consisted of 1251 lines, most of them translated automatically from the SCR specification, and theoretically had an execution space of 1.3×10^{22} states. In their model checking, the researchers found that one of the properties that was thought not to hold did indeed hold: it turned out that the conditions under which it would not hold were unreachable. They also discovered that a safety property did not hold: according to the specification, it was possible for the weapon delivery system to use stale data in tracking a target's location when the navigation sensors were deemed to be reporting unreasonable values.

A **theorem prover** uses a collection of built-in theories, inference rules, and decision procedures for determining whether a set of asserted facts logically entails some unasserted fact; most sophisticated theorem provers require human assistance in sketching out the proof strategy. Dutertre and Stavridou (1997) used the theorem prover PVS to verify some of the functional and safety requirements of an avionics system. For example, the assumption that relates the wing sweep angle WSPOS at time *t + eps* and the wing sweep command CMD at time *t,* in the case where none of the interlocks is active, is expressed in PVS as:

```
cmd_wings : AXIOM
  constant_in_interval (CMD, t, t + eps)
and
  not wings_locked_in_interval (t, t + eps)
implies
  CMD (t) = WSPOS (t + eps)
or
  CMD (t) < WSPOS (t + eps) and
  WSPOS (t + eps) <= WPOS (t) - eps * ws_min_rate
or
  CMD (t) > WSPOS (t + eps) and
  WSPOS (t + eps) >= WPOS (t) - eps * ws_min_rate
```

The entire PVS model, including specification and assumptions, consisted of about 4500 lines of PVS, including comments and blank lines. The verification of requirements involved two steps. Some theorems were proved to check that the PVS model was internally consistent and complete, and others were used directly to prove three main safety properties. The proof of safety was quite complex. In total, 385 proofs were performed, of which about 100 were discharged automatically by the theorem prover, and the rest required human guidance. It took approximately 6 person-months to write the PVS model and support libraries, and another 12 person-months to formalize the assumptions and carry out the verification; the verification of the three main safety properties took

4.10 Measuring Requirements

There are many ways to measure characteristics of requirements, so that the information collected tells us a lot about the requirements process and about the quality of the requirements themselves. Measurements usually focus on three areas: product, process and resources (Fenton and Pfleeger 1997). , The number of requirements in the requirements definition and specification can give us a sense of how large the developed system is likely to be. We saw in Chapter 3 that effort-estimation models require an estimate of product size, and requirements size can be used as input to such models. Moreover, requirements size, and effort estimation, can be tracked throughout development. As design and development lead to a deeper understanding of both problem and solution, new requirements may arise that were not apparent during the initial requirements-capture process.

Similarly, we can measure the number of changes to requirements. A large number of changes indicates some instability or uncertainty in our understanding of what the system should do or how it should behave, and suggests that we should take actions to try lower the rate of changes. This tracking of changes also can continue throughout development; as the system requirements change, the impact of those changes can be assessed.

Where possible, requirements-size and change measurements should be recorded by requirements type. Such category-based metrics tell us whether change or uncertainty in requirements is product wide, or rests solely with certain kinds of requirements, such as user-interface or database requirements. This information helps us to determine if we need to focus our attention on particular types of requirements.

Because the requirements are used by the designers and testers, we may want to devise measures that reflect their assessment of the requirements. For example, we can ask the designers to rate each requirement on a scale from 1 to 5:

1: You (the designer) understand this requirement completely, you have designed from similar requirements in the past, and you should have no trouble developing a design from this requirement.

2: There are elements of this requirement that are new to you, but they are not radically different from requirements you have successfully designed from in the past.

3: There are elements of this requirement that are very different from requirements you have designed from in the past, but you understand the requirement and think you can develop a good design from it.

4: There are parts of this requirement that you do not understand, and you are not sure that you can develop a good design.

5: You do not understand this requirement at all, and you cannot develop a design for it.

We can create a similar rating scheme that asks testers how well they understand each requirement and how confident they are in being able to devise a suitable test suite for each requirement. In both cases, the profiles of rankings can serve as a coarse indicator of whether the requirements are written at the appropriate level of detail. If the designers and testers yield profiles with mostly 1s and 2s, as shown in Figure 4.28(a), then the requirements are in good shape and can be passed on to the design team. However, if there are many 4s and 5s, as shown in Figure 4.28(b), then the requirements should be revised, and the revisions reassessed to have better profiles, before we proceed to design. Although the assessment is subjective, the general trends should be clear, and the scores can provide useful feedback to both us and our customers.

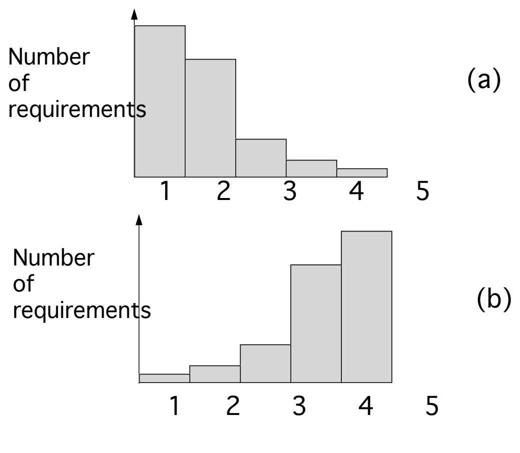


FIGURE 4.28 Measuring requirements readiness.

We can also take note, for each requirement, of when it is reviewed, implemented as design, implemented as code, and tested. These measures tell us the progress we are making toward completion. Testers can also measure the thoroughness of their test cases with respect to the requirements, as we will see in Chapters 8 and 9. We can measure the number of requirements covered by each test case, and the number of requirements that have been tested (Wilson 1995).

4.11 CHOOSING A SPECIFICATION TECHNIQUE

This chapter has presented examples of several requirements specification techniques, and many more are available for use on your projects. Each one has useful characteristics, but some are more appropriate for a given project than others. That is, no technique is best for all projects. Thus, it is important to have a set of criteria for deciding, for each project, which technique is most suitable.

Let us consider some of the issues that should be included in such a set of criteria. Suppose we are to build a computerized system for avoiding collisions among aircraft. Participating aircraft are to be fitted with radar sensors. A subsystem on each aircraft is to monitor other aircraft in its vicinity, detect when an aircraft is flying dangerously close, establish communications with that aircraft, negotiate evasive maneuvers to avoid a collision (after all, we wouldn't want both planes to independently choose maneuvers that would put them or keep them on a collision course!), and instruct its navigation system to execute the negotiated maneuvers. Each aircraft's subsystem performs its own data analysis and decision-making procedures on onboard computers, although it shares flight plans with other aircraft and transmits all data and final maneuvers to a central site for further analysis. One of the key characteristics of this collision-avoidance system is that it is a distributed, reactive system. That is, it is a **reactive system** in that each aircraft's subsystem is continuously monitoring and reacting to the positions of other aircraft. It is a **distributed system** in that the system's functions are distributed over several aircraft. The complexity of this system makes it essential that the requirements be specified exactly and completely. Interfaces must be well-defined, and communications must be coordinated, so that each aircraft's subsystem can make decisions in a timely manner. Some specification techniques may be more appropriate than others for this problem. For example, testing this system will be difficult because, for safety reasons, most testing cannot take place in the real environment. Moreover, it will be hard to detect and replicate transient errors. Thus, we might prefer a technique that offers simulation, or that facilitates exhaustive or automated verification of the specification. In particular, techniques that automatically check the specification or system for consistency and completeness may catch errors that are not easy to spot otherwise.

More generally, if a system has real-time requirements, we need a specification technique that supports the notion of time. Any need for phased development means that we will be tracking requirements through several intermediate systems – which not only complicates the requirements tracking, but also increases the likelihood that the requirements will change over the life of the system. As the users work with intermediate versions of the system, they may see the need for new features, or want to change existing features. Thus, we need a sophisticated method that can handle change easily. If we want our requirements to have all of the desirable characteristics listed early in the chapter, then we look for a method that helps us to revise the requirements, track the changes, cross-reference the data and functional items, and analyze the requirements for as many characteristics as possible.

Ardis and his colleagues (Ardis et al. 1996) have proposed a set of criteria for evaluating specification methods. They associate with each criterion a list of questions to help us to determine how well a particular method satisfies that criterion. These criteria were intended for evaluating techniques for specifying reactive systems, but as you will see, most of the criteria are quite general:

- **Applicability:** Can the technique describe real-world problems and solutions in a natural and realistic way? If the technique makes assumptions about the environment, are the assumptions reasonable? Is the technique compatible with the other techniques that will be used on the project?
- **Implementability:** Can the specification be refined or translated easily into an implementation? How difficult is the translation? Is it automated? If so, is the generated code efficient? Is the generated code in the same language as is used in the manually produced parts of the implementation? Is there a clean, well-defined interface between the code that is machine-generated and the code that is not?

- **Testability/simulation:** Can the specification be used to test the implementation? Is every statement in the specification testable by the implementation? Is it possible to execute the specification?
- **Checkability:** Are the specifications readable by nondevelopers, such as the customer)? Can domain experts (i.e., experts on the problem being specified) check the specification for accuracy? Are there automated specification checkers?
- **Maintainability:** Will the specification be useful in making changes to the system? Is it easy to change the specification as the system evolves?
- **Modularity:** Does the method allow a large specification to be decomposed into smaller parts that are easier to write and to understand? Can changes be made to the smaller parts without rewriting the entire specification?
- **Level of abstraction/expressibility:** How closely and expressively do objects, states, events in the language correspond to the actual objects, actions, and conditions in the problem domain? How concise and elegant is the resulting specification?
- **Soundness:** Does the language or do the tools facilitate checking for inconsistencies or ambiguities in the specification? Are the semantics of the specification language defined precisely?
- **Verifiability:** Can we demonstrate formally that the specification satisfies the requirements? Can the verification process be automated, and, if so, is the automation easy?
- **Run-time safety:** If code can be generated automatically from the specification, does the code degrade gracefully under unexpected run-time conditions, such as overflow?
- **Tools maturity:** If the specification technique has tool support, are the tools of high quality? Is there training available for learning how to use them? How large is the user base for the tools?
- **Looseness:** Can the specification be incomplete or admit nondeterminism?
- **Learning curve:** Can a new user learn quickly the technique's concepts, syntax, semantics, and heuristics?
- **Technique maturity**: Has the technique been certified or standardized? Is there a user group or large user base?
- **Data modeling:** Does the technique include data representation, relationships, or abstractions? Are the data-modeling facilities an integrated part of the technique?
- **Discipline:** Does the technique force its users to write well-structured, understandable, and well-behaved specifications?

The first step in choosing a specification technique is to determine for our particular problem which of the above criteria are especially important. Different problems place different priorities on the criteria. Ardis and his colleagues were interested in developing telephone switching systems, and so they judged whether each of the criteria is helpful in developing reactive systems. They considered not only the criteria's effects on requirements activities, but their effects on other life-cycle activities as well. Table 4.4 shows the results of their evaluation. The second step in choosing a specification technique is to evaluate each of the candidate techniques with respect to the criteria. For example, Ardis and colleagues rated Z as strong in modularity, abstraction, verifiability, looseness, technique maturity, and data modeling; adequate in applicability, checkability, maintainability, soundness, tools maturity, learning curve, and discipline; and weak in implementability and testability/simulation. Some of their assessments of Z, such as Z inherently supports modularity, hold for all problem types, whereas other assessments, such as applicability, are specific to the problem type. In the end, we choose a specification technique that best supports the criteria that are most importance to our particular problem.

Table 4.4 Importance of Specification Criteria During Reactive-system Life-cycle (Ardis et al. 1996)
(R=Requirements, D=Design, I=Implementation, T=Testing, M=Maintenance, O=Other) © 1996 IEEE

R	**D**	**I**	**T**	**M**	**O**	**Criteria**
+		+				Applicability
		+		+		Implementability
+	+		+			Testability/simulation
+			+	+		Checkability
				+		Maintainability
	+			+		Modularity
+	+					Level of abstraction/expressability
+			+			Soundness
+	+	+	+	+		Verifiability
		+		+		Run-time safety
		+	+	+		Tools maturity
+						Looseness
					+	Learning curve
					+	Technique maturity
	+					Data modeling
+	+	+		+		Discipline

Since no one approach is universally applicable to all systems, it may be necessary to combine several approaches to define the requirements completely. Some methods are better at capturing control flow and synchronization, whereas other methods are better at capturing data transformations. Some problems are more easily described in terms of events and actions, whereas other problems are better described in terms of control states that reflect stages of behavior. Thus, it may be useful to use one method for data requirements and another to describe processes or time-related activities. We may need to express changes in behavior as well as global invariants. Models that are adequate for designers may be difficult for the test team to use. Thus, the choice of a specification technique(s) is bound up in the characteristics of the individual project and the preferences of developers and customers.

4.12 INFORMATION SYSTEMS EXAMPLE

Recall that our Piccadilly example involves selling advertising time for the Piccadilly Television franchise area. We can use several specification notations to model the requirements related to buying and selling advertising time. Because this problem is an information system, we will use only notations that are data-oriented.

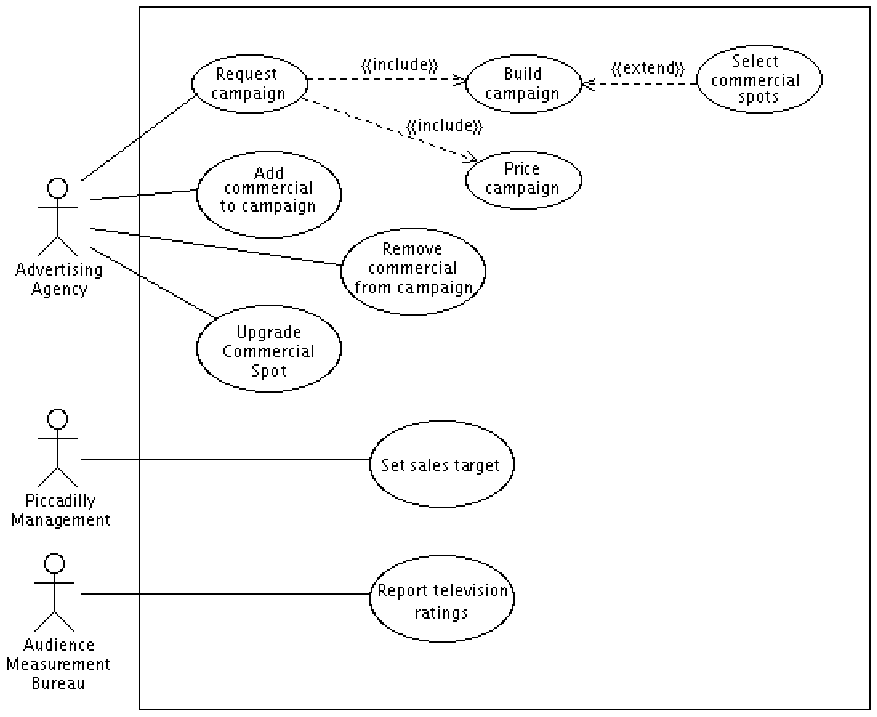


Figure 4.29 Use case for the Piccadilly Television advertising system (adapted from Robertson and Robertson 1994).

First, we can draw a use-case diagram to represent the key uses of the system, showing the expected users and the major functions that each user might initiate. A partial diagram might look like Figure 4.29. Notice that this high-level diagram captures the essential functionality of the system, but it shows nothing about the ways in which each of these use cases might succeed or fail; for example, a campaign request would fail if all of the commercial time were already sold. The diagram also shows nothing about the type of information that the system might input, process, or output. We need more information about each of the use, to better understand the problem.

As a next step, we can draw event traces, such as the one shown in Figure 4.30, that depict typical scenarios within a use case. For example, the request for a campaign involves

- Searching each relevant commercial break to see whether there is any unsold time and whether commercials during that break are likely to be seen by the campaign's intended target audience
- Computing the price of the campaign, based on the prices of the available commercial spots found
- Reserving available commercial spots for the campaign

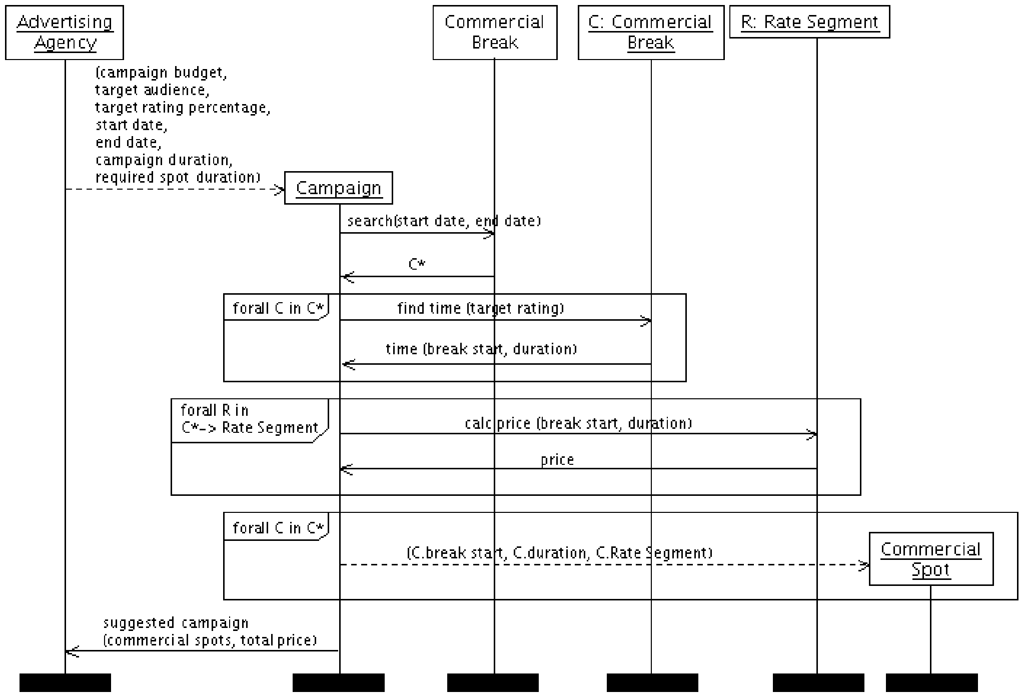

Advertising Agency
Commercial Break
C: Commercial Break
R: Rate Segment
(campaign budget, target audience, target rating percentage, start date, end date, campaign duration, required spot duration)
Campaign
search(start date, end date)
C*
forall C in C*
find time (target rating)
time (break start, duration)
forall R in C*-> Rate Segment
calc price (break start, duration)
price
forall C in C*
(C.break start, C.duration, C.Rate Segment)
Commercial Spot
suggested campaign (commercial spots, total price)

Figure 4.30 Message Sequence Chart for a successful request for an advertising campaign (adapted from Robertson and Robertson 1994).

Figure 4.30 uses UML-style Message Sequence Charts, in which entities whose names are underlined represent object instances, whereas entities whose names are not underlined represent abstract classes. Thus, the search for available commercial spots is done by first asking the class for the set C* of relevant Commercial Breaks, and then asking each of these Commercial Break instances whether it has available time that is suitable for the campaign. Boxes surround sequences of messages that are repeated for multiple instances: for example, reserving time in multiple Commercial Break instances, or creating multiple Commercial Spots. The resulting campaign of commercial spots and the campaign's price are returned to the requesting Advertising Agency. Similar traces can be drawn for other possible responses to a campaign request and for other use cases. In drawing these traces, we start to identify key entities and relationships, which we can record in a UML class diagram, as in Figure 4.31.

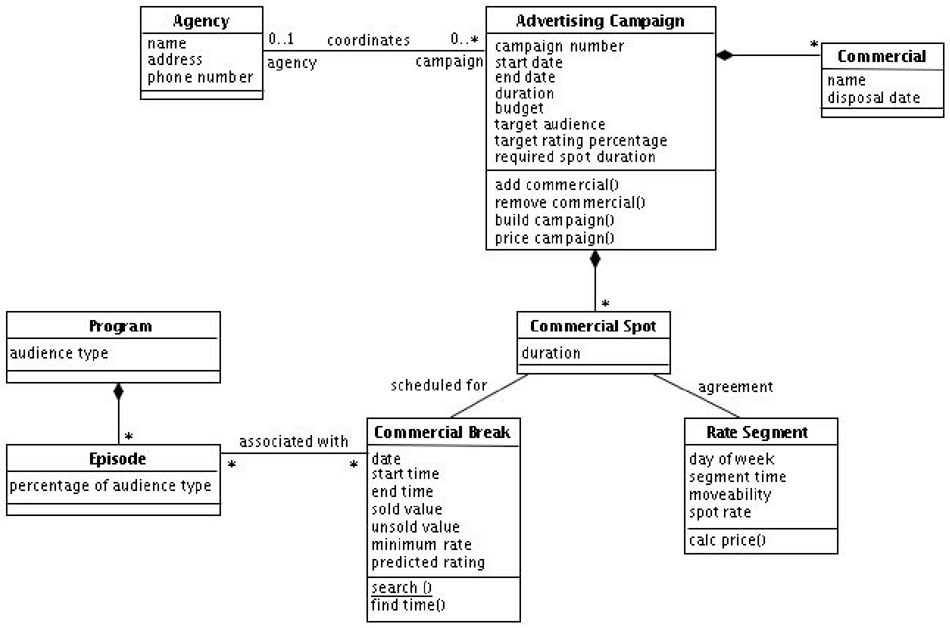


Figure 4.31 Partial UML class diagram of Piccadilly Television advertising system (adapted from Robertson and Robertson 1994).

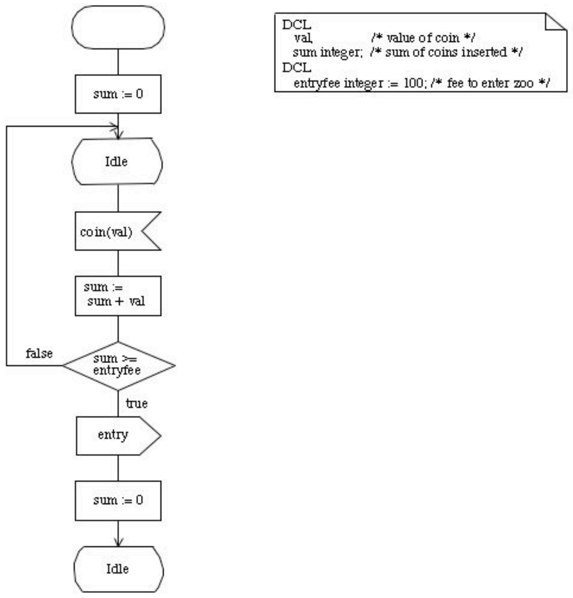
The complete specification for Piccadilly is quite long and involved, and the Robertsons' book provides many of the details. However, the examples here make it clear that different notations are suitable for representing different aspects of a problem's requirements; it is important to choose a combination of techniques that paints a complete picture of the problem, to be used in designing, implementing, and testing the system.

4.13 REAL-TIME EXAMPLE

Recall that the Ariane-5 explosion was caused by the reuse of a section of code from Ariane-4. Nuseibeh (1997) analyzes the problem from the point of view of requirements reuse. That is, many software engineers feel that great benefits can be had from reusing requirements specifications (and their related design, code, and test cases) from previously developed systems. Candidate specifications are identified by looking for functionality or behavioral requirements that are the same or similar, and then making modifications where necessary. In the case of Ariane-4, the inertial reference system (SRI) performed many of the functions needed by Ariane-5.

However, Nuseibeh notes that although the needed functionality was similar to that in Ariane-4, there were aspects of Ariane-5 that were significantly different. In particular, the SRI functionality that continued after liftoff in Ariane-4 was not needed after liftoff in Ariane-5. Had requirements validation been done properly, the analysts would have discovered that the functions active after liftoff could not be traced back to any Ariane-5 requirement in the requirements definition or specification. Thus, requirements validation could have played a crucial role in preventing the rocket's explosion.

Another preventive measure might have been to simulate the requirements. Simulation would have shown that the SRI continued to function after liftoff; then, Ariane-5's design could have been changed to reuse a modified version of the SRI code. Consider again the list of criteria proposed by Ardis and colleagues for selecting a specification language. This list includes two items that are especially important for specifying a system such as Ariane-5: testability/simulation and run-time safety. In Ardis's study, the team examined seven specification languages – Modechart, VFSM, Esterel, Lotos, Z, SDL, and C – for suitability against each of the criteria; only SDL was rated "strong" for testability/simulation and run-time safety. An SDL model consists of several concurrent communicating processes like the Coin Slot process in Figure 4.32.



```
DCL
  val,            /* value of coin */
  sum integer;  /* sum of coins inserted */
DCL
  entryfee integer := 100; /* fee to enter zoo */
```

Figure 4.32 SDL process for the coin slot of the turnstile problem.

To validate an SDL model, the system requirements can be written as temporal-logic invariants:

```
CLAIM;
  Barrier = locked
  IMPLIES (Barrier = locked)
      UNLESS (sum >= entryfee);
ENDCLAIM;
```

SDL is a mature formal method that includes object-oriented concepts and powerful modeling features: for example, processes can be spawned dynamically and be assigned identifiers, events can be directed to specific processes by referring to process identifiers, processes can store persistent data, events can carry data parameters, and timers can model real-time delays and deadlines. Commercial tools are available to support design, debugging, and maintenance of SDL specifications. Thus, one possible prevention technique might have been the use of a specification method like SDL, with accompanying tool support.

We will see in later chapters that preventive steps could also have been taken during design, implementation, or testing; however, measures taken during requirements analysis would have led to a greater understanding of the differences between Ariane-4 and Ariane-5, and to detecting the root cause of the error.

WHAT THIS CHAPTER MEANS FOR YOU

In this chapter, we have shown the best practices for developing quality software requirements. We have seen that requirements activities should not be performed by the software developer in isolation: definition and specification efforts require working closely with users, customers, testers, designers, and other team members. Still, there are several skills that are important for you to master on your own:

- It is essential that the requirements definition and specification documents describe the problem, leaving solution selection to the designers. The best way of ensuring that you do not stray into the solution space is to describe requirements and specifications in terms of environmental phenomena.

- There are a variety of sources and means for eliciting requirements. There are both functional and quality requirements to keep in mind. The functional requirements explain what the system will do, and the quality requirements constrain solutions in terms of safety, reliability, budget, schedule, and so on.
- There are many different types of definition and specification techniques. Some are descriptive, such as entity-relationship diagrams and logic, while others are behavioral, such as event traces, data-flow diagrams, and functions. Some have graphical notations, and some are based on mathematics. Each emphasizes a different view of the problem, and suggests different criteria for decomposing a problem into subproblems. It is often desirable to use a combination of techniques to specify the different aspects of a system.
- The specification techniques also differ in terms of their tool support, maturity, understandability, ease of use, and mathematical formality. Each one should be judged for the project at hand, as there is no best, universal technique.
- Requirements questions can be answered using models or prototypes. In either case, the goal is to focus on the subproblem that is at the heart of the question, rather than necessarily modeling or prototyping the entire problem. If prototyping, you need to decide ahead of time whether the resulting software will be kept or thrown away.
- Requirements must be validated to ensure that they accurately reflect the customer's expectations. The requirements should also be checked for completeness, correctness, consistency, feasibility, and more, sometimes using techniques or tools that are associated with the specification methods you have chosen. Finally, you should verify that the specification fulfills the requirements.

WHAT THIS CHAPTER MEANS FOR YOUR DEVELOPMENT TEAM

Your development team must work together to elicit, understand, and document requirements. Often, different team members concentrate on separate aspects of the requirements: the networking expert may work on network requirements, the user-interface expert on screens and reports, the database expert on data capture and storage, and so on. Because the disparate requirements will be integrated into a comprehensive whole, requirements must be written in a way that allows them to be linked and controlled. For example, a change to one requirement may affect other, related requirements, and the methods and tools must support the changes to ensure that errors are caught early and quickly.

At the same time, the requirements part of your team must work closely with

- Customers and users, so that your team builds a product that serves their needs
- Designers, so that they construct a design that fulfills the requirements specification
- Testers, so that their test scripts adequately evaluate whether the implementation meets the requirements
- Documentation writers, so that they can write user manuals from the specifications

Your team must also pay attention to measurements that reflect requirements quality. The measures can suggest team activities, such as prototyping some requirements when indicators show that the requirements are not well-understood.

Finally, you must work as a team to review the requirements definition and specification documents, and to update those documents as the requirements change and grow during the development and maintenance processes.

WHAT THIS CHAPTER MEANS FOR RESEARCHERS

There are many research areas associated with requirements activities. Researchers can

- Investigate ways to reduce the amount of uncertainty and risk in requirements

- Develop specification techniques and tools that permit easier ways to prove assumptions and assertions, and to demonstrate consistency, completeness, and determinism
- Develop tools to allow traceability across the various intermediate and final products of software development. In particular, the tools can assess the impact of a proposed change on products, processes, and resources.
- Evaluate the many different ways to review requirements: tools, checklists, inspections, walk-throughs, and more. It is important to know which techniques are best for what situations.
- Create new techniques for simulating requirements behavior.
- Help us to understand what types of requirements are best for reuse in subsequent projects, and how to write requirements in a way that enhances their later reuse.

4.17 TERM PROJECT

Your clients at FCO have prepared the following set of English-language requirements for the Loan Arranger system. Like most sets of requirements, this set must be scrutinized in several ways to determine if it is correct, complete and consistent. Using the requirements here and in supplementary material about the Loan Arranger in earlier chapters, evaluate and improve this set of requirements. Use many of the techniques presented in this chapter, including requirements measurement and Ardis' list. If necessary, express the requirements in a requirements language or modeling technique, to make sure that the static and dynamic properties of the system are expressed well.

Preconditions and Assumptions

- The Loan Arranger system assumes that there already exist lenders, borrowers and loans from which to choose, and that investors exist who are interested in buying bundles of loans.
- The Loan Arranger system contains a repository of information about loans from a variety of lenders. This repository may be empty.
- At regular intervals, each lender provides reports listing the loans that it has made. Loans that have already been purchased by FCO will be indicated on these reports.
- Each loan in the Loan Arranger repository represents an investment to then be bundled and sold with other loans.
- The Loan Arranger system may be used by up to four loan analysts simultaneously.

High-Level Description of Functionality

1. The Loan Arranger system will receive monthly reports from each lender of new loans issued by that lender. The loans in the report recently purchased by FCO for its investment portfolio will be marked in the report. The Loan Arranger system will use the report information to update its repository of available loans.
2. The Loan Arranger system will receive monthly reports from each lender providing updates about the status of loans issued by that lender. The updated information will include: the current interest rate for an adjustable rate mortgage, and the status of the borrower with respect to the loan (good, late or default). For loans in the FCO portfolio, the Loan Arranger will update the data in the repository. Loans not in the FCO portfolio will also be examined in order to determine if a borrower's standing should be updated. FCO will provide each lender with the format for the reports, so that all reports will share a common format.
3. The loan analyst can change individual data records as described in Data Operations.
4. All new data must be validated before they are added to the repository (according to the rules described in Data Constraints).
5. The loan analyst can use Loan Arranger to identify bundles of loans to sell to particular investors.

Functional Requirements

1. The loan analyst should be able to review all of the information in the repository for a particular lending institution, a particular loan, or a particular borrower.
2. The loan analyst can create, view, edit or delete a loan from a portfolio or bundle.
3. A loan is added to the portfolio automatically, when the Loan Arranger reads the reports provided by the lenders. A report can be read by the Loan Arranger only after the associated lender has been specified.
4. The loan analyst can create a new lender.
5. The loan analyst can delete a lender only if there are no loans in the portfolio associated with this lender.
6. The loan analyst can change lender contact and phone number but not lender name and identification number.
7. The loan analyst cannot change borrower information.
8. The loan analyst can ask the system to sort, search or organize loan information by certain criteria: amount, interest rate, settlement date, borrower, lender, type of loan, or whether it has been marked for inclusion in a certain bundle. The organizational criteria should include ranges, so that information will be included only if it is within two specified bounds (such as between January 1, 1999 and January 1, 2002). The organizational criteria can also include exclusion, such as all loans not marked, or all loans not between January 1, 1999 and January 1, 2002.
9. The loan analyst should be able to request reports in each of three formats: a file, on the screen, and as a printed report.
10. The loan analyst should be able to request the following information in a report: any attribute of loan, lender or borrower, and summary statistics of the attributes (mean, standard deviation, scatter diagram and histogram). The information in a report can be restricted to a subset of the total information, as described by the loan analyst's organizing criteria.
11. The loan analyst must be able to use the Loan Arranger to create bundles that meet the prescribed characteristics of an investment request. The loan analyst can identify these bundles in several ways:
    - by manually identifying a subset of loans that must be included in the bundle, either by naming particular loans or by describing them using attributes or ranges
    - by providing the Loan Arranger with the investment criteria, and allowing the Loan Arranger to run a loan bundle optimization request to select the best set of loans to meet those criteria
    - by using a combination of the above, where a subset of loans is first chosen (manually or automatically) and then optimizing the chosen subset according to the investment criteria
12. Creating a bundle consists of two steps. First, the loan analyst works with the Loan Arranger to create a bundle according to criteria, as described above. Then the candidate bundle can be accepted, rejected or modified. Modifying a bundle means that the analyst may accepted some but not all of the loans suggested by the Loan Arranger for a bundle, and can add specific loans to the bundle before accepting it.
13. The loan analyst must be able to mark loans for possible inclusion in a loan bundle. Once a loan is so marked, it is not available for inclusion in any other bundle. If the loan analyst marks a loan and decides not to include it in the bundle, the marking must be removed and the loan made available for other bundling decisions.
14. When a candidate bundle is accepted, its loans are removed from consideration for use in other bundles.
15. All current transactions must be resolved before a loan analyst can exit the Loan Arranger system.
16. A loan analyst can access a repository of investment requests. This repository may be empty. For each investment request, the analyst uses the request constraints (on risk, profit and term) to define the parameters of a bundle. Then, the Loan Arranger system identifies loans to be bundled to meet the request constraints.

Data Constraints

1. A single borrower may have more than one loan.
2. Every lender must have a unique identifier.

3. Every borrower must have a unique identifier.
4. Each loan must have at least one borrower.
5. Each loan must have a loan amount of at least $1000 but not more than $500,000.
6. There are two types of loans based on the amount of the loan: regular and jumbo. A regular loan is for any amount less than or equal to $275,000. A jumbo loan is for any amount over $275,000.
7. A borrower is considered to be in good standing if all loans to that borrower are in good standing. A borrower is considered to be in default standing if any of the loans to that borrower have default standing. A borrower is said to be in late standing if any of the loans to that borrower have late standing.
8. A loan or borrower can change from good to late, from good to default, from late to good, or from late to default. Once a loan or borrower is in default standing, it cannot be changed to another standing.
9. A loan can change from ARM to FM, and from FM to ARM.
10. The profit requested by an investor is a number from 0 to 500. 0 represents no profit on a bundle. A non-zero profit represents the rate of return on the bundle; if the profit is x, then the investor expects to receive the original investment plus x percent of the original investment when the loans are paid off. Thus, if a bundle costs $1000, and the investor expects a rate of return of 40, then the investor hopes to have $1400 when all loans in the bundle are paid off.
11. No loan can appear in more than one bundle.

Design and Interface Constraints

1. The Loan Arranger system should work on a Unix system.
2. The loan analyst should be able to look at information about more than one loan, lending institution, or borrower at a time.
3. The loan analyst must be able to move forward and backwards through the information presented on a screen. When the information is too voluminous to fit on a single screen, the user must be informed that more information can be viewed.
4. When the system displays the results of a search, the current organizing criteria must always be displayed along with the information.
5. A single record or line of output must never be broken in the middle of a field.
6. The user must be advised when a search request is inappropriate or illegal.
7. When an error is encountered, the system should return the user to the previous screen.

Quality Requirements:

1. Up to four loan analysts can use the system at a given time.
2. If updates are made to any displayed information, the information is refreshed within five seconds of adding, updating or deleting information.
3. The system must respond to a loan analyst's request for information in less than five seconds from submission of the request.
4. The system must be available for use by a loan analyst during 97% of the business day.

4.18 KEY REFERENCES

Michael Jackson's book (1995) on *Software Requirements and Specifications* provides general advice on how to overcome common problems in understanding and formulating requirements. His ideas can be applied to any requirements technique. Donald Gause and Gerald Weinberg's book (1989) on *Exploring Requirements* focuses on the human side of the requirements process: problems and techniques for working with customers and users and for devising new products.

A comprehensive requirements-definition template developed by James and Suzanne Robertson can be found at the Web site of the Atlantic Systems Guild: http://www.systemsguild.com. This template is accompanied by a description of the Volere process model, which is a complete process for eliciting and checking a set of requirements. Use of the template is described in their book *Mastering the Requirements Process* (1999).

Peter Coad and Edward Yourdon's book (1991), *Object-oriented Analysis*, is a classic text on object-oriented requirements analysis. The most thorough references on the Unified Modeling Language (UML) are the books by James Rumbaugh, Ivan Jacobson, and Grady Booch, especially the *Unified Modeling Language Reference Manual*, and the documents released by the Object Management Group; the latter can be downloaded from the organization's Web site: http://www.omg.org. Martin Fowler's book (1996) on *Analysis Patterns* provides guidance on how to model in UML common business problems.

Since 1993, the IEEE Computer Society has sponsored two conferences that are directly related to requirements and that have been held in alternate years: the International Conference on Requirements Engineering and the International Symposium on Requirements Engineering. These conference merged in 2002 to form the International Requirements Engineering Conference, which is held every year. Information about upcoming conferences and about proceedings from past conferences can be found at the Computer Society's Web page: http://www.computer.org.

The *Requirements Engineering Journal* focuses exclusively on new results in eliciting, representing, and validating requirements, mostly with respect to software systems. *IEEE Software* has had special issues on requirements engineering, in March 1994, March 1996, March/April 1998, May/June 2000, January/February 2003, and March/April 2004. Other IEEE publications often have special issues on particular types of requirements analysis and specification methods. For example, the September 1990 issues of *IEEE Computer*, *IEEE Software*, and *IEEE Transactions on Software Engineering* focused on formal methods, as did the May 1997 and January 1998 issues of *IEEE Transactions on Software Engineering* and the April 1996 issue of *IEEE Computer.*

There are several standards related to software requirements. The U.S. Department of Defense has produced MilStd-498, Data Item Description for Software Requirements Specifications (SRS). The IEEE has produced IEEE Std 830-1998, which is a set of recommended practices and standards for formulating and structuring requirements specifications.

There are several tools that support requirements capture and traceability. DOORS/ERS (Telelogic), Analyst Pro (Goda Software), and RequisitePro (IBM Rational) are popular tools for managing requirements, tracing requirements in downstream artifacts, tracking changes, and assessing the impact of changes. Most modeling notations have tool support that at the least supports the creation and editing of models, usually supports some form of well-formedness checking and report generation, and at the best offers automated validation and verification. An independent survey of requirements tools is located at www.systemsguild.com.

There is an IFIP Working Group 2.9 on Software Requirements Engineering. Some of the presentations from their annual meeting are available from their Web site: http://www.cis.gsu.edu/~wrobinso/ifip2_9

4.19 EXERCISES

1. Developers work together with customers and users to define requirements and specify what the proposed system will do. If, once it is built, the system works according to specification but harms someone physically or financially, who is responsible?

2. Among the many nonfunctional requirements that can be included in a specification are those related to safety and reliability. How can we ensure that these requirements are testable, in the sense defined by the Robertsons? In particular, how can we demonstrate the reliability of a system that is required never to fail?

3. In an early meeting with your customer, the customer lists the following "requirements" for a system he wants you to build:

a) The client daemon must be invisible to the user

b) The system should provide automatic verification of corrupted links or outdated data

c) An internal naming convention should ensure that records are unique

d) Communication between the database and servers should be encrypted

e) Relationships may exist between title groups [a type of record in the database]

f) Files should be organizable into groups of file dependencies

g) The system must interface with an Oracle database

h) The system must handle 50,000 users concurrently

Classify each of the above as a functional requirement, a quality requirement, a design constraint, or a process constraint. Which of the above might be premature design decisions? Re-express each of these decisions as a requirement that the design decision was meant to achieve.

4. Write a decision table that specifies the rules for the game of checkers.

5. If a decision table has two identical columns, then the requirements specification is redundant. How can we tell if the specification is contradictory? What other characteristics of a decision table warn us of problems with the requirements?

6. Write a Parnas table that describes the output of the algorithm for finding the roots of a quadratic equation using the quadratic formula.

7. Write a state-machine specification to illustrate the requirements of an automatic banking machine (ABM).

8. A state-machine specification is **complete** if and only if there is a transition specified for every possible combination of state and input symbol. We can change an incomplete specification to a complete one by adding an extra state, called a trap state. Once a transition is made to the trap state, the system remains in the trap state, no matter the input. For example, if 0, 1, and 2 are the only possible inputs, the system depicted by Figure 4.33 can be completed by adding a trap state as shown in Figure 4.34. In same manner, complete your state-machine specification from Exercise 6.

(old Figure 4.25 from 2nd edition)

Figure 4.33 Original system for Exercise 7.

(old Figure 4.26 from 2nd edition)

Figure 4.34 Complete system with trap state for Exercise 7.

9. A **safety property** is an invariant property that specifies that a particular bad behavior never happens; for example, a safety property of the turnstile problem is that the number of entries into the zoo is never more than the number of entry fees paid. A **liveness property** is a property that specifies that a particular behavior eventually happens; for example, a liveness property for the turnstile problem is that when an entry fee is paid, the turnstile becomes unlocked. Similarly, a liveness property for the library system is that every borrow request from a Patron who has no outstanding library fines succeeds. These three properties, when expressed in logic, look like the following:

$\Box\,(\text{num\_coins} \geq \text{num\_entries})$

$\Box\,(\text{insert\_coin} \Rightarrow \bigcirc\ \text{barrier=unlocked})$

$\Box\,(\ (\text{borrow(Patron,Pub)} \wedge \text{Patron.fines} = 0) \Rightarrow$

$\qquad \Diamond\ \exists\ \text{Loan.}\ [\text{Loan.borrower=Patron} \wedge \text{Loan.Publication} = \text{Pub}]\ )$

List safety and liveness properties for your automated banking machine specification from Exercise 6. Express these properties in temporal logic.

10. Prove that your safety and liveness properties from Exercise 8 hold for your state-machine model of your automated banking machine specification from Exercise 6. What assumptions do you have to make about the ABM's environment (e.g., that the machine has sufficient cash) for your proofs to succeed?

11. Sometimes part of a system may be built quickly to demonstrate feasibility or functionality to a customer. This prototype system is usually incomplete; the real system is constructed after the customer and developer evaluate the prototype. Should the system requirements document be written before or after a prototype is developed? Why?

12. Write a set of UML models (use-case diagram, MSC diagrams, class diagram) for an on-line telephone directory to replace the phonebook that is provided to you by your phone company. The directory should be able to provide phone numbers when presented with a name; it should also list area codes for different parts of the country and generate emergency telephone numbers for your area.

13. Draw data-flow diagrams to illustrate the functions and data flow for the on-line telephone directory system specified in the previous problem.

14. What are the benefits of separating functional flow from data flow?

15. What special kinds of problems are presented when specifying the requirements of real-time systems?

16. Contrast the benefits of an object-oriented requirements specification with those of a functional decomposition.

17. Write a Z specification for a presentation scheduling system. The system keeps a record of which presenters are to give presentations on which dates. No presenter should be scheduled to give more than one presentation. No more than four presentations should be scheduled for any particular date. There should be operations to Add and Remove presentations from the schedule, to Swap the dates of two presentations, to List the presentations scheduled for a particular date, to List the date on which a particular presenter is scheduled to speak, and to send a Reminder message to each presenter on the date of his or her presentation. You may define any additional operations that help simplify the specification.

18. Complete the partial SDL data specification for the library problem in Figure 4.20. In particular, write axioms for nongenerator operations Unreserve, isOnLoan, and isOnReserve. Modify your axioms for operation Unreserve so that this operation assumes that multiple requests to put an item on Reserve might occur between two requests to Unreserve that item.

19. What kinds of problems should you look for when doing a requirements review? Make a checklist of these problems. Can the checklist be universally applicable or is it better to use a checklist that is specific to the application domain?

20. Is it ever possible to have the requirements definition document be the same as the requirements specification? What are the pros and cons of having two documents?

21. Pfleeger and Hatton (1997) examined the quality of a system that had been specified using formal methods. They found that the system was unusually well-structured and easy to test. They speculated that the high quality was due to the thoroughness of the specification, not necessarily its formality. How could you design a study to determine whether it is formality or thoroughness that leads to high quality?

22. Sometimes a customer requests a requirement that you know is impossible to implement. Should you agree to put the requirement in the definition and specification documents anyway, thinking that you might come up with a novel way of meeting it, or thinking that you will ask that the requirement be dropped later? Discuss the ethical implications of promising what you know you cannot deliver.

23. Find a set of natural-language requirements at your job or at this bookÆ’s web site. Review the requirements to determine if there are any problems. For example, are they consistent? Ambiguous? Conflicting? Do they contain any design or implementation decisions? Which representation techniques might help reveal and eliminate these problems? If the problems remain in the requirements, what is their likely impact as the system is designed and implemented?