CHAPTER

10 SYSTEM ENGINEERING

KEY CONCEPTS

Almost 500 years ago, Machiavelli said: "there is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success, than to take the lead in the introduction of a new order of things." During the past 50 years, computer-based systems have introduced a new order. Although technology has made great strides since Machiavelli spoke, his words continue to ring true.

Software engineering occurs as a consequence of a process called *system engineering*. Instead of concentrating solely on software, system engineering focuses on a variety of elements, analyzing, designing, and organizing those elements into a system that can be a product, a service, or a technology for the transformation of information or control.

The system engineering process is called *business process engineering* when the context of the engineering work focuses on a business enterprise. When a product (in this context, a product includes everything from a wireless telephone to an air traffic control system) is to be built, the process is called *product engineering*.

Both business process engineering and product engineering attempt to bring order to the development of computer-based systems. Although each is applied in a different application domain, both strive to put software into context. That

QUICK LOOK

What is it? Before software can be engineered, the "system" in which it resides must be understood. To accomplish this, the overall objective of the system must be determined; the role of hardware, software, people, database, procedures, and other system elements must be identified; and operational requirements must be elicited, analyzed, specified, modeled, validated, and managed. These activities are the foundation of system engineering.

Who does it? A system engineer works to understand system requirements by working with the customer, future users, and other stakeholders.

Why is it important? There's an old saying: "You can't see the forest for the trees." In this context, the "forest" is the system, and the trees are the technology elements (including software) that are required to realize the system. If you rush to build technology elements before you understand the system, you'll undoubtedly make mistakes that will disappoint your customer. Before you worry about the trees, understand the forest.

What are the steps? Objectives and more detailed operational requirements are identified by eliciting information from the customer; requirements are analyzed to assess their clarity, completeness, and consistency; a specification, often incorporating a system model, is created and then validated by both practitioners and customers. Finally, system requirements are managed to ensure that changes are properly controlled.

QUICK LOOK

What is the work product? An effective representation of the system must be produced as a consequence of system engineering. This can be a prototype, a specification or even a symbolic model, but it must communicate the operational, functional, and behavioral characteristics of the system to be built and provide insight into the system architecture.

How do I ensure that I've done it right? Perform requirements engineering steps, including requirements elicitation, that lead to a solid specification. Then review all system engineering work products for clarity, completeness, and consistency. As important, expect changes to the system requirements and manage them using solid SCM (Chapter 9) methods.

is, both business process engineering and product engineering[1] work to allocate a role for computer software and, at the same time, to establish the links that tie software to other elements of a computer-based system.

In this chapter, we focus on the management issues and the process-specific activities that enable a software organization to ensure that it does the right things at the right time in the right way.

10.1 COMPUTER-BASED SYSTEMS

The word *system* is possibly the most overused and abused term in the technical lexicon. We speak of political systems and educational systems, of avionics systems and manufacturing systems, of banking systems and subway systems. The word tells us little. We use the adjective describing system to understand the context in which the word is used. *Webster's Dictionary* defines *system* in the following way:

> 1. a set or arrangement of things so related as to form a unity or organic whole; 2. a set of facts, principles, rules, etc., classified and arranged in an orderly form so as to show a logical plan linking the various parts; 3. a method or plan of classification or arrangement; 4. an established way of doing something; method; procedure . . .

Five additional definitions are provided in the dictionary, yet no precise synonym is suggested. *System* is a special word.

Borrowing from Webster's definition, we define a *computer-based system* as

> A set or arrangement of elements that are organized to accomplish some predefined goal by processing information.

The goal may be to support some business function or to develop a product that can be sold to generate business revenue. To accomplish the goal, a computer-based system makes use of a variety of system elements:

---

1 In reality, the term *system engineering* is often used in this context. However, in this book, the term *system engineering* is generic and is used to encompass both business process engineering and product engineering.

Software. Computer programs, data structures, and related documentation that serve to effect the logical method, procedure, or control that is required.
Hardware. Electronic devices that provide computing capability, the interconnectivity devices (e.g., network switches, telecommunications devices) that enable the flow of data, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function.
People. Users and operators of hardware and software.
Database. A large, organized collection of information that is accessed via software.
Documentation. Descriptive information (e.g., hardcopy manuals, on-line help files, Web sites) that portrays the use and/or operation of the system.
Procedures. The steps that define the specific use of each system element or the procedural context in which the system resides.

Don't be lured into taking a "software-centric" view. Begin by considering all elements of a system before you concentrate on software.

The elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product; a robot transforms a command file containing specific instructions into a set of control signals that cause some specific physical action. Creating an information system to assist the marketing department and control software to support the robot both require system engineering.

One complicating characteristic of computer-based systems is that the elements constituting one system may also represent one macro element of a still larger system. The macro element is a computer-based system that is one part of a larger computer-based system. As an example, we consider a "factory automation system" that is essentially a hierarchy of systems. At the lowest level of the hierarchy we have a numerical control machine, robots, and data entry devices. Each is a computer-based system in its own right. The elements of the numerical control machine include electronic and electromechanical hardware (e.g., processor and memory, motors, sensors), software (for communications, machine control, interpolation), people (the machine operator), a database (the stored NC program), documentation, and procedures. A similar decomposition could be applied to the robot and data entry device. Each is a computer-based system.

Complex systems are actually a hierarchy of macro elements that are themselves systems.

At the next level in the hierarchy, a manufacturing cell is defined. The manufacturing cell is a computer-based system that may have elements of its own (e.g., computers, mechanical fixtures) and also integrates the macro elements that we have called numerical control machine, robot, and data entry device.

To summarize, the manufacturing cell and its macro elements each are composed of system elements with the generic labels: software, hardware, people, database, procedures, and documentation. In some cases, macro elements may share a generic element. For example, the robot and the NC machine both might be managed by a single operator (the people element). In other cases, generic elements are exclusive to one system.

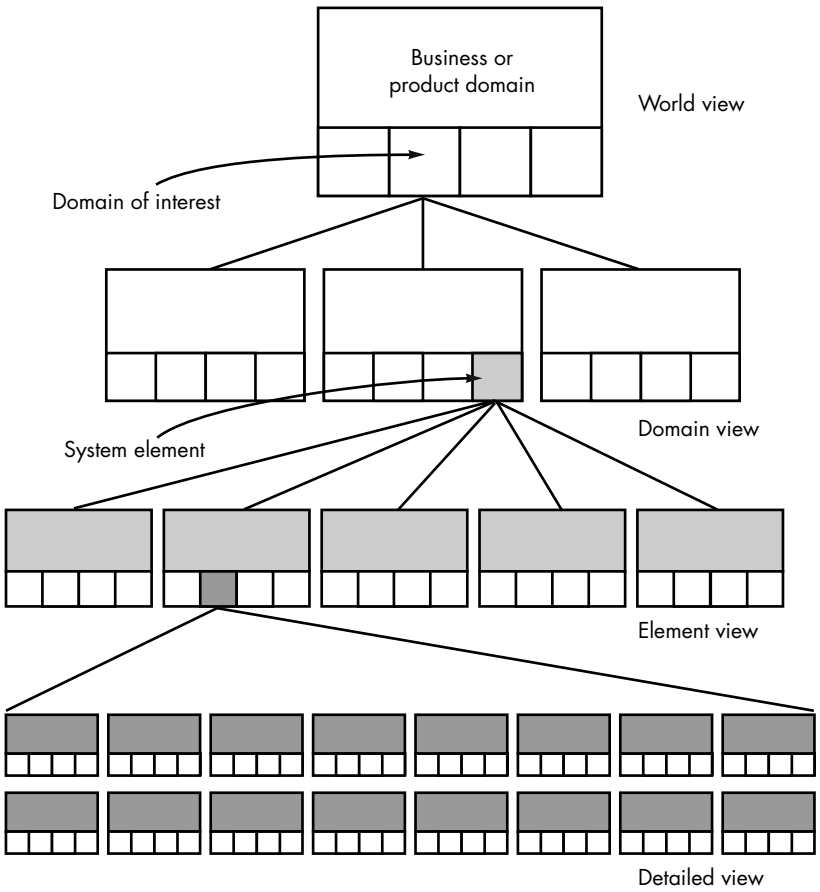


FIGURE 10.1
The system engineering hierarchy

The role of the system engineer is to define the elements for a specific computer-based system in the context of the overall hierarchy of systems (macro elements). In the sections that follow, we examine the tasks that constitute computer system engineering.

10.2 THE SYSTEM ENGINEERING HIERARCHY

Regardless of its domain of focus, system engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy illustrated in Figure 10.1. The system engineering process usually begins with a "world view." That is, the entire business or product domain is examined to ensure that the proper business or technology context can be established. The world view is refined to focus more fully on specific domain of interest. Within a specific domain, the need for targeted system elements (e.g., data, software, hardware, people) is analyzed. Finally, the analysis,

design, and construction of a targeted system element is initiated. At the top of the hierarchy, a very broad context is established and, at the bottom, detailed technical activities, performed by the relevant engineering discipline (e.g., hardware or software engineering), are conducted.[2]

Stated in a slightly more formal manner, the world view (WV) is composed of a set of domains (D_i), which can each be a system or system of systems in its own right.

$$\text{WV} = \{D_1, D_2, D_3, \ldots, D_n\}$$

KEY POINT

Good system engineering begins with a clear understanding of context—the world view— and then progressively narrows focus until technical detail is understood.

Each domain is composed of specific elements (E_j) each of which serves some role in accomplishing the objective and goals of the domain or component:

$$D_i = \{E_1, E_2, E_3, \ldots, E_m\}$$

Finally, each element is implemented by specifying the technical components (C_k) that achieve the necessary function for an element:

$$E_j = \{C_1, C_2, C_3, \ldots, C_k\}$$

In the software context, a component could be a computer program, a reusable program component, a module, a class or object, or even a programming language statement.

It is important to note that the system engineer narrows the focus of work as he or she moves downward in the hierarchy just described. However, the world view portrays a clear definition of overall functionality that will enable the engineer to understand the domain, and ultimately the system or product, in the proper context.

10.2.1 System Modeling

System engineering is a modeling process. Whether the focus is on the world view or the detailed view, the engineer creates models that [MOT92]

? What does a system engineering model accomplish?

- Define the processes that serve the needs of the view under consideration.
- Represent the behavior of the processes and the assumptions on which the behavior is based.
- Explicitly define both exogenous and endogenous input[3] to the model.
- Represent all linkages (including output) that will enable the engineer to better understand the view.

To construct a system model, the engineer should consider a number of restraining factors:

---

2 In some situations, however, system engineers must first consider individual system elements and/or detailed requirements. Using this approach, subsystems are described bottom up by first considering constituent detailed components of the subsystem.

3 *Exogenous* inputs link one constituent of a given view with other constituents at the same level or other levels; *endogenous* input links individual components of a constituent at a particular view.

1. *Assumptions* that reduce the number of possible permutations and variations, thus enabling a model to reflect the problem in a reasonable manner. As an example, consider a three-dimensional rendering product used by the entertainment industry to create realistic animation. One domain of the product enables the representation of 3D human forms. Input to this domain encompasses the ability to specify movement from a live human actor, from video, or by the creation of graphical models. The system engineer makes certain assumptions about the range of allowable human movement (e.g., legs cannot be wrapped around the torso) so that the range of inputs and processing can be limited.
2. *Simplifications* that enable the model to be created in a timely manner. To illustrate, consider an office products company that sells and services a broad range of copiers, faxes, and related equipment. The system engineer is modeling the needs of the service organization and is working to understand the flow of information that spawns a service order. Although a service order can be derived from many origins, the engineer categorizes only two sources: internal demand and external request. This enables a simplified partitioning of input that is required to generate the service order.
3. *Limitations* that help to bound the system. For example, an aircraft avionics system is being modeled for a next generation aircraft. Since the aircraft will be a two-engine design, the monitoring domain for propulsion will be modeled to accommodate a maximum of two engines and associated redundant systems.
4. *Constraints* that will guide the manner in which the model is created and the approach taken when the model is implemented. For example, the technology infrastructure for the three-dimensional rendering system described previously is a single G4-based processor. The computational complexity of problems must be constrained to fit within the processing bounds imposed by the processor.
5. *Preferences* that indicate the preferred architecture for all data, functions, and technology. The preferred solution sometimes comes into conflict with other restraining factors. Yet, customer satisfaction is often predicated on the degree to which the preferred approach is realized.

A system engineer considers the following factors when developing alternative solutions: assumptions, simplifications, limitations, constraints, and customer preferences.

The resultant system model (at any view) may call for a completely automated solution, a semi-automated solution, or a nonautomated approach. In fact, it is often possible to characterize models of each type that serve as alternative solutions to the problem at hand. In essence, the system engineer simply modifies the relative influence of different system elements (people, hardware, software) to derive models of each type.

10.2.2 System Simulation

In the late 1960s, R. M. Graham [GRA69] made a distressing comment about the way we build computer-based systems: "We build systems like the Wright brothers built airplanes—build the whole thing, push it off a cliff, let it crash, and start over again." In fact, for at least one class of system—the *reactive system*—we continue to do this today.

If simulation capability is unavailable for a reactive system, project risk increases. Consider using an iterative process model that will enable you to deliver a working product in the first iteration and then use other iterations to tune its performance.

Many computer-based systems interact with the real world in a reactive fashion. That is, real-world events are monitored by the hardware and software that form the computer-based system, and based on these events, the system imposes control on the machines, processes, and even people who cause the events to occur. Real-time and embedded systems often fall into the reactive systems category.

Unfortunately, the developers of reactive systems sometimes struggle to make them perform properly. Until recently, it has been difficult to predict the performance, efficiency, and behavior of such systems prior to building them. In a very real sense, the construction of many real-time systems was an adventure in "flying." Surprises (most of them unpleasant) were not discovered until the system was built and "pushed off a cliff." If the system crashed due to incorrect function, inappropriate behavior, or poor performance, we picked up the pieces and started over again.

Many systems in the reactive category control machines and/or processes (e.g., commercial aircraft or petroleum refineries) that must operate with an extremely high degree of reliability. If the system fails, significant economic or human loss could occur. For this reason, the approach described by Graham is both painful and dangerous.

Today, software tools for system modeling and simulation are being used to help to eliminate surprises when reactive, computer-based systems are built. These tools are applied during the system engineering process, while the role of hardware and software, databases and people is being specified. Modeling and simulation tools enable a system engineer to "test drive" a specification of the system. The technical details and specialized modeling techniques that are used to enable a test drive are discussed briefly in Chapter 31.

CASE Tools
Modeling & Simulation

10.3 BUSINESS PROCESS ENGINEERING: AN OVERVIEW

The goal of business process engineering (BPE) is to define architectures that will enable a business to use information effectively. Michael Guttman [GUT99] describes the challenge when he states:

> [T]oday's computing environment consists of computing power that's distributed over an enterprise-wide array of heterogeneous processing units, scaled and configured for a wide variety of tasks. Variously known as client-server computing, distributed processing, and enterprise networking (to name just a few overused terms), this new environment promised businesses the greater functionality and flexibility they demanded.

> However, the price for this change is largely borne by the IT [information technology] organizations that must support this polyglot configuration. Today, each IT organization must become, in effect, its own systems integrator and architect. It must design, implement, and support its own unique configuration of heterogeneous computing resources, distributed logically and geographically throughout the enterprise, and connected by an appropriate enterprise-wide networking scheme.
>
> Moreover, this configuration can be expected to change continuously, but unevenly, across the enterprise, due to changes in business requirements and in computing technology. These diverse and incremental changes must be coordinated across a distributed environment consisting of hardware and software supplied by dozens, if not hundreds, of vendors. And, of course, we expect these changes to be seamlessly incorporated without disrupting normal operations and to scale gracefully as those operations expand.

When taking a world view of a company's information technology needs, there is little doubt that system engineering is required. Not only is the specification of the appropriate computing architecture required, but the software architecture that populates the "unique configuration of heterogeneous computing resources" must be developed. Business process engineering is one approach for creating an overall plan for implementing the computing architecture [SPE93].

Three different architectures are developed during BPE: data architecture, application architecture, and technology infrastructure.

Three different architectures must be analyzed and designed within the context of business objectives and goals:

- data architecture
- applications architecture
- technology infrastructure

XRef

Data objects are discussed in detail in Chapter 12.

The *data architecture* provides a framework for the information needs of a business or business function. The individual building blocks of the architecture are the data objects that are used by the business. A data object contains a set of attributes that define some aspect, quality, characteristic, or descriptor of the data that are being described. For example, an information engineer might define the data object **customer.** To more fully describe **customer,** the following attributes are defined:

Object: Customer
Attributes:
name
company name
job classification and purchase authority
business address and contact information
product interest(s)
past purchase(s)
date of last contact
status of contact

Once a set of data objects is defined, their relationships are identified. A *relationship* indicates how objects are connected to one another. As an example, consider the

objects: **customer,** and **product A.** The two objects can be connected by the relationship *purchases;* that is, a customer purchases product A or product A is purchased by a customer. The data objects (there may be hundreds or even thousands for a major business activity) flow between business functions, are organized within a database, and are transformed to provide information that serves the needs of the business.

XRef

A detailed discussion of software architecture is presented in Chapter 14.

The *application architecture* encompasses those elements of a system that transform objects within the data architecture for some business purpose. In the context of this book, we consider the application architecture to be the system of programs (software) that performs this transformation. However, in a broader context, the application architecture might incorporate the role of people (who are information transformers and users) and business procedures that have not been automated.

The *technology infrastructure* provides the foundation for the data and application architectures. The infrastructure encompasses the hardware and software that are used to support the application and data. This includes computers, operating systems, networks, telecommunication links, storage technologies, and the architecture (e.g., client/server) that has been designed to implement these technologies.

As a software engineer, you may never get involved in ISP or BAA. However, if it's clear that these activities haven't been done, inform the stakeholders that the project risk is very high.

To model the system architectures described earlier, a hierarchy of business process engineering activities is defined. Referring to Figure 10.2, the world view is achieved through *information strategy planning* (ISP). ISP views the entire business as an entity and isolates the domains of the business (e.g., engineering, manufacturing, marketing, finance, sales) that are important to the overall enterprise. ISP defines the data objects that are visible at the enterprise level, their relationships, and how they flow between the business domains [MAR90].

The domain view is addressed with a BPE activity called *business area analysis* (BAA). Hares [HAR93] describes BAA in the following manner:

> BAA is concerned with identifying in detail data (in the form of entity [data object] types) and function requirements (in the form of processes) of selected business areas [domains] identified during ISP and ascertaining their interactions (in the form of matrices). It is only concerned with specifying what is required in a business area.

Business Process Engineering

As the system engineer begins BAA, the focus narrows to a specific business domain. BAA views the business area as an entity and isolates the business functions and procedures that enable the business area to meet its objectives and goals. BAA, like ISP, defines data objects, their relationships, and how data flow. But at this level, these characteristics are all bounded by the business area being analyzed. The outcome of BAA is to isolate areas of opportunity in which information systems may support the business area.

Once an information system has been isolated for further development, BPE makes a transition into software engineering. By invoking a *business system design* (BSD) step, the basic requirements of a specific information system are modeled and these requirements are translated into data architecture, applications architecture, and technology infrastructure.

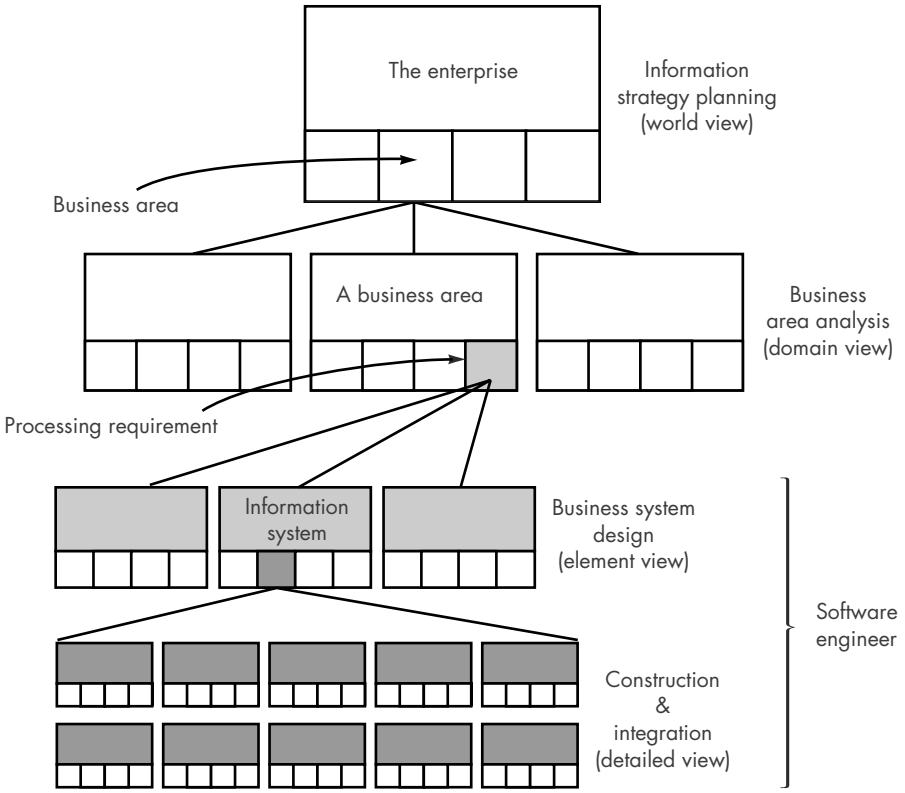


FIGURE 10.2 The business process engineering hierarchy

The final BPE step—*construction and integration* focuses on implementation detail. The architecture and infrastructure are implemented by constructing an appropriate database and internal data structures, by building applications using software components, and by selecting appropriate elements of a technology infrastructure to support the design created during BSD. Each of these system components must then be integrated to form a complete information system or application. The integration activity also places the new information system into the business area context, performing all user training and logistics support to achieve a smooth transition.[4]

10.4 PRODUCT ENGINEERING: AN OVERVIEW

The goal of product engineering is to translate the customer's desire for a set of defined capabilities into a working product. To achieve this goal, product engineering—like

4 It should be noted that the terminology (adapted from [MAR90]) used in Figure 10.2 is associated with information engineering, the predecessor of modern BPE. However, the area of focus implied by each activity noted is addressed by all who consider the subject.

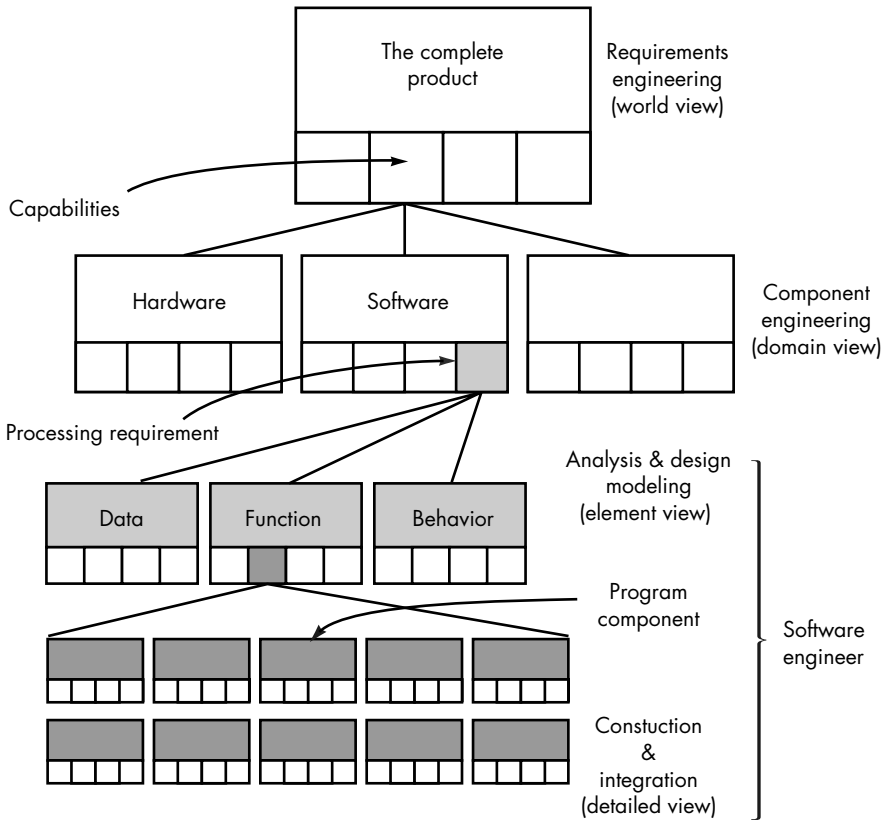


FIGURE 10.3
The product engineering hierarchy

business process engineering—must derive architecture and infrastructure. The architecture encompasses four distinct system components: software, hardware, data (and databases), and people. A support infrastructure is established and includes the technology required to tie the components together and the information (e.g., documents, CD-ROM, video) that is used to support the components.

Referring to Figure 10.3, the world view is achieved through *requirements engineering.* The overall requirements of the product are elicited from the customer. These requirements encompass information and control needs, product function and behavior, overall product performance, design and interfacing constraints, and other special needs. Once these requirements are known, the job of requirements engineering is to allocate function and behavior to each of the four components noted earlier.

Once allocation has occurred, *system component engineering* commences. System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering. Each of these engineering disciplines takes a domain-specific view, but it is important to note that the engineering disciplines must establish and maintain active communication with one another. Part of

the role of requirements engineering is to establish the interfacing mechanisms that will enable this to happen.

The element view for product engineering is the engineering discipline itself applied to the allocated component. For software engineering, this means *analysis and design modeling activities* (covered in detail in later chapters) and *construction and integration activities* that encompass code generation, testing, and support steps. The analysis step models allocated requirements into representations of data, function, and behavior. Design maps the analysis model into data, architectural, interface, and software component-level designs.

10.5 REQUIREMENTS ENGINEERING

Quote:

"The hardest single part of building a software system is deciding what to build. . . . No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Fred Brooks

The outcome of the system engineering process is the specification of a computer-based system or product at the different levels described generically in Figure 10.1. But the challenge facing system engineers (and software engineers) is profound: How can we ensure that we have specified a system that properly meets the customer's needs and satisfies the customer's expectations? There is no foolproof answer to this difficult question, but a solid requirements engineering process is the best solution we currently have.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system [THA97]. The requirements engineering process can be described in five distinct steps [SOM97]:

- requirements elicitation
- requirements analysis and negotiation
- requirements specification
- system modeling
- requirements validation
- requirements management

WebRef

A detailed report entitled "Issues in Requirements Elicitation" can be downloaded from **www.sei.cmu.edu/publications/documents/92.reports/92.tr.012.html**

10.5.1 Requirements Elicitation

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis. But it isn't simple—it's very hard.

Christel and Kang [CRI92] identify a number of problems that help us understand why requirements elicitation is difficult:

? Why is it so difficult to gain a clear understanding of what the customer wants?

- *Problems of scope*. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- *Problems of understanding.* The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or untestable.
- *Problems of volatility.* The requirements change over time.

To help overcome these problems, system engineers must approach the requirements gathering activity in an organized manner.

Sommerville and Sawyer [SOM97] suggest a set of detailed guidelines for requirements elicitation, which are summarized in the following steps:

Be sure you've assessed overall feasibility before you expend effort and time eliciting detailed requirements.

- Assess the business and technical feasibility for the proposed system.
- Identify the people who will help specify requirements and understand their organizational bias.
- Define the technical environment (e.g., computing architecture, operating system, telecommunications needs) into which the system or product will be placed.
- Identify "domain constraints" (i.e., characteristics of the business environment specific to the application domain) that limit the functionality or performance of the system or product to be built.
- Define one or more requirements elicitation methods (e.g., interviews, focus groups, team meetings).
- Solicit participation from many people so that requirements are defined from different points of view; be sure to identify the rationale for each requirement that is recorded.
- Identify ambiguous requirements as candidates for prototyping.
- Create usage scenarios (see Chapter 11) to help customers/users better identify key requirements.

XRef

Requirements elicitation methods are presented in Chapter 11.

The work products produced as a consequence of the requirements elicitation activity will vary depending on the size of the system or product to be built. For most systems, the work products include

- A statement of need and feasibility.
- A bounded statement of scope for the system or product.

- A list of customers, users, and other stakeholders who participated in the requirements elicitation activity.
- A description of the system's technical environment.
- A list of requirements (preferably organized by function) and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in the requirements elicitation.

10.5.2 Requirements Analysis and Negotiation

Once requirements have been gathered, the work products noted earlier form the basis for *requirements analysis*. Analysis categorizes requirements and organizes them into related subsets; explores each requirement in relationship to others; examines requirements for consistency, omissions, and ambiguity; and ranks requirements based on the needs of customers/users.

What questions must be asked and answered during requirements analysis?

As the requirements analysis activity commences, the following questions are asked and answered:

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?

Requirements Analysis

It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources. It also is relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."

If different customers/users cannot agree on requirements, the risk of failure is very high. Proceed with extreme caution.

The system engineer must reconcile these conflicts through a process of negotiation. Customers, users and stakeholders are asked to rank requirements and then discuss conflicts in priority. Risks associated with each requirement are identified and analyzed (see Chapter 6 for details). Rough guestimates of development effort are made and used to assess the impact of each requirement on project cost and delivery time. Using an iterative approach, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

10.5.3 Requirements Specification

In the context of computer-based systems (and software), the term *specification* means different things to different people. A specification can be a written document, a graphical model, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Negotiation Techniques

Some suggest that a "standard template" [SOM97] should be developed and used for a system specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner. However, it is sometimes necessary to remain flexible when a specification is to be developed. For large systems, a written document, combining natural language descriptions and graphical models may be the best approach. However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

System Specification

The *System Specification* is the final work product produced by the system and requirements engineer. It serves as the foundation for hardware engineering, software engineering, database engineering, and human engineering. It describes the function and performance of a computer-based system and the constraints that will govern its development. The specification bounds each allocated system element. The *System Specification* also describes the information (data and control) that is input to and output from the system.

10.5.4 System Modeling

Assume for a moment that you have been asked to specify all requirements for the construction of a gourmet kitchen. You know the dimensions of the room, the location of doors and windows, and the available wall space. You could specify all cabinets and appliances and even indicate where they are to reside in the kitchen. Would this be a useful specification?

The answer is obvious. In order to fully specify what is to be built, you would need a meaningful model of the kitchen, that is, a blueprint or three-dimensional rendering that shows the position of the cabinets and appliances and their relationship to one another. From the model, it would be relatively easy to assess the efficiency of work flow (a requirement for all kitchens), the aesthetic "look" of the room (a personal, but very important requirement).

We build system models for much the same reason that we would develop a blueprint or 3D rendering for the kitchen. It is important to evaluate the system's components in relationship to one another, to determine how requirements fit into this picture, and to assess the "aesthetics" of the system as it has been conceived. Further discussion of system modeling is presented in Section 10.6.

10.5.5 Requirements Validation

The work products produced as a consequence of requirements engineering (a system specification and related information) are assessed for quality during a validation step. *Requirements validation* examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

A key concern during requirements validation is consistency. Use the system model to ensure that requirements have been consistently stated.

The primary requirements validation mechanism is the formal technical review (Chapter 8). The review team includes system engineers, customers, users, and other stakeholders who examine the system specification[5] looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

Although the requirements validation review can be conducted in any manner that results in the discovery of requirements errors, it is useful to examine each requirement against a set of checklist questions. The following questions represent a small subset of those that might be asked:

Requirements

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called *validation criteria*) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?

---

5 In reality, many FTRs are conducted as the system specification is developed. It is best for the review team to examine small portions of the specification, so that attention can be focused on a specific aspect of the requirements.

- Is the system specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with system performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Checklist questions like these help ensure that the validation team has done everything possible to conduct a thorough review of each requirement.

An article entitled "Making Requirements Management Work for You" contains pragmatic guidelines: **stsc.hill.af.mil/crosstalk/1999/apr/davis.asp**

10.5.6 Requirements Management

In the preceding chapter, we noted that requirements for computer-based systems change and that the desire to change requirements persists throughout the life of the system. *Requirements management* is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management techniques discussed in Chapter 9.

Like SCM, requirements management begins with identification. Each requirement is assigned a unique identifier that might take the form

<requirement type><requirement #>

Many requirements management activities are borrowed from SCM.

where requirement type takes on values such as *F* = functional requirement, *D* = data requirement, *B* = behavioral requirement, *I* = interface requirement, and *P* = output requirement. Hence, a requirement identified as F09 indicates a functional requirement assigned requirement number 9.

Once requirements have been identified, traceability tables are developed. Shown schematically in Figure 10.4, each traceability table relates identified requirements to one or more aspects of the system or its environment. Among many possible traceability tables are the following:

When a system is large and complex, determining the "connections" among requirements can be a daunting task. Use traceability tables to make the job a bit easier.

Features traceability table. Shows how requirements relate to important customer observable system/product features.
Source traceability table. Identifies the source of each requirement.
Dependency traceability table. Indicates how requirements are related to one another.
Subsystem traceability table. Categorizes requirements by the subsystem(s) that they govern.
Interface traceability table. Shows how requirements relate to both internal and external system interfaces.

In many cases, these traceability tables are maintained as part of a requirements database so that they may be quickly searched to understand how a change in one requirement will affect different aspects of the system to be built.

FIGURE 10.4 Generic traceability table

Requirement \ Specific aspect of the system or its environment	A01	A02	A03	A04	A05				Aii
R01			✓		✓				
R02	✓		✓						
R03	✓			✓					✓
R04		✓			✓				
R05	✓	✓		✓					✓
Rnn	✓		✓						

10.6 SYSTEM MODELING

Every computer-based system can be modeled as an information transform using an input-processing-output template. Hatley and Pirbhai [HAT87] have extended this view to include two additional system features—user interface processing and maintenance and self-test processing. Although these additional features are not present for every computer-based system, they are very common, and their specification makes any system model more robust.

XRef

Other system modeling methods take an object-oriented view. The UML approach can be applied at the system level and is discussed in Chapters 21 and 22.

Using a representation of input, processing, output, user interface processing, and self-test processing, a system engineer can create a model of system components that sets a foundation for later steps in each of the engineering disciplines.

To develop the system model, a *system model template* [HAT87] is used. The system engineer allocates system elements to each of five processing regions within the template: (1) user interface, (2) input, (3) system function and control, (4) output, and (5) maintenance and self-test. The format of the architecture template is shown in Figure 10.5.

Like nearly all modeling techniques used in system and software engineering, the system model template enables the analyst to create a hierarchy of detail. A *system context diagram* (SCD) resides at the top level of the hierarchy. The context diagram "establishes the information boundary between the system being implemented and the environment in which the system is to operate" [HAT87]. That is, the SCD defines all external producers of information used by the system, all external consumers of information created by the system, and all entities that communicate through the interface or perform maintenance and self-test.

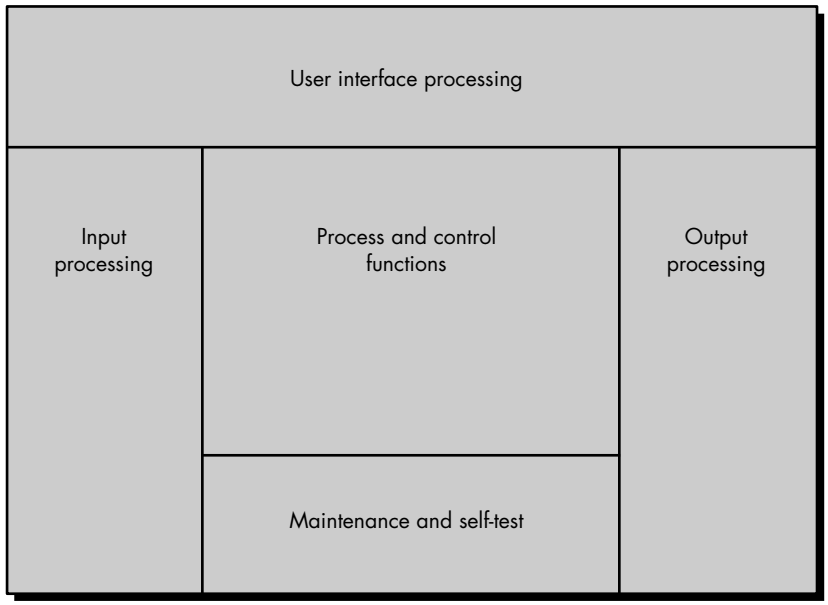


FIGURE 10.5
System model template [HAT87]

To illustrate the use of the SCD, consider the conveyor line sorting system that was introduced in Chapter 5. The system engineer is presented with the following (somewhat nebulous) statement of objectives for CLSS:

> CLSS must be developed such that boxes moving along a conveyor line will be identified and sorted into one of six bins at the end of the line. The boxes will pass by a sorting station where they will be identified. Based on an identification number printed on the side of the box (an equivalent bar code is provided), the boxes will be shunted into the appropriate bins. Boxes pass in random order and are evenly spaced. The line is moving slowly.

The SCD provides a "big picture" view of the system you must build. Every detail need not be specified at this level. Refine the SCD hierarchically to elaborate the system.

For this example, CLSS is extended and makes use of a personal computer at the sorting station site. The PC executes all CLSS software, interacts with the bar code reader to read part numbers on each box, interacts with the conveyor line monitoring equipment to acquire conveyor line speed, stores all part numbers sorted, interacts with a sorting station operator to produce a variety of reports and diagnostics, sends control signals to the shunting hardware to sort the boxes, and communicates with a central factory automation mainframe. The SCD for CLSS (extended) is shown in Figure 10.6.

Each box shown in Figure 10.6 represents an *external entity*—that is, a producer or consumer of system information. For example, the bar code reader produces information that is input to the CLSS system. The symbol for the entire system (or, at lower levels, major subsystems) is a rectangle with rounded corners. Hence, CLSS is represented in the processing and control region at the center of the SCD. The labeled

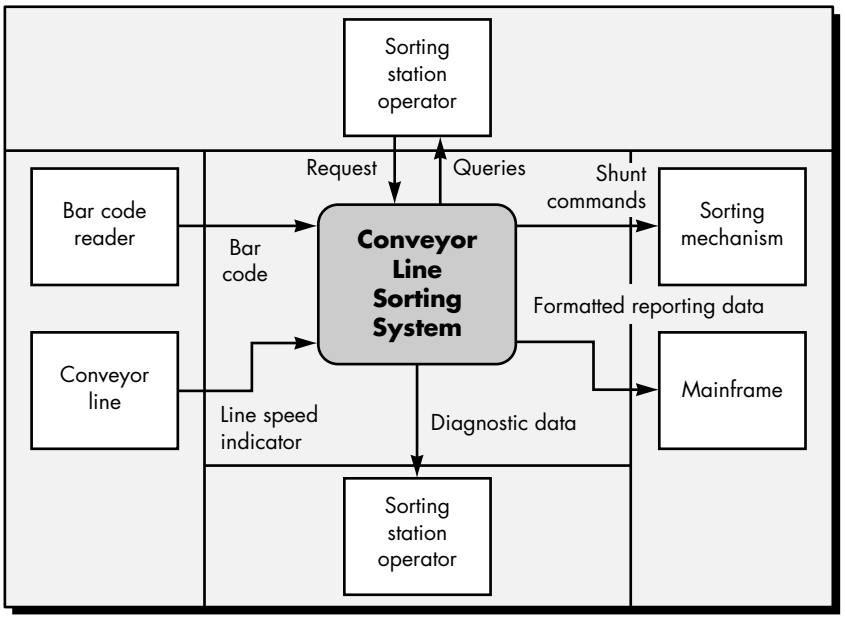


FIGURE 10.6
System context diagram for CLSS (extended)

arrows shown in the SCD represent information (data and control) as it moves from the external environment into the CLSS system. The external entity *bar code reader* produces input information that is labeled **bar code**. In essence, the SCD places any system into the context of its external environment.

The system engineer refines the system context diagram by considering the shaded rectangle in Figure 10.6 in more detail. The major subsystems that enable the conveyor line sorting system to function within the context defined by the SCD are identified. Referring to Figure 10.7, the major subsystems are defined in a *system flow diagram* (SFD) that is derived from the SCD. Information flow across the regions of the SCD is used to guide the system engineer in developing the SFD—a more detailed "schematic" for CLSS. The system flow diagram shows major subsystems and important lines of information (data and control) flow. In addition, the system template partitions the subsystem processing into each of the five regions discussed earlier. At this stage, each of the subsystems can contain one or more system elements (e.g., hardware, software, people) as allocated by the system engineer.

XRef
The SFD is a precursor to the data flow diagram, discussed in Chapter 12.

The initial system flow diagram becomes the top node of a hierarchy of SFDs. Each rounded rectangle in the original SFD can be expanded into another architecture template dedicated solely to it. This process is illustrated schematically in Figure 10.8. Each of the SFDs for the system can be used as a starting point for subsequent engineering steps for the subsystem that has been described.

WebRef
A useful white paper on Hatley-Pirbhai method can be found at **www.hasys.com/papers/hp_description.html**

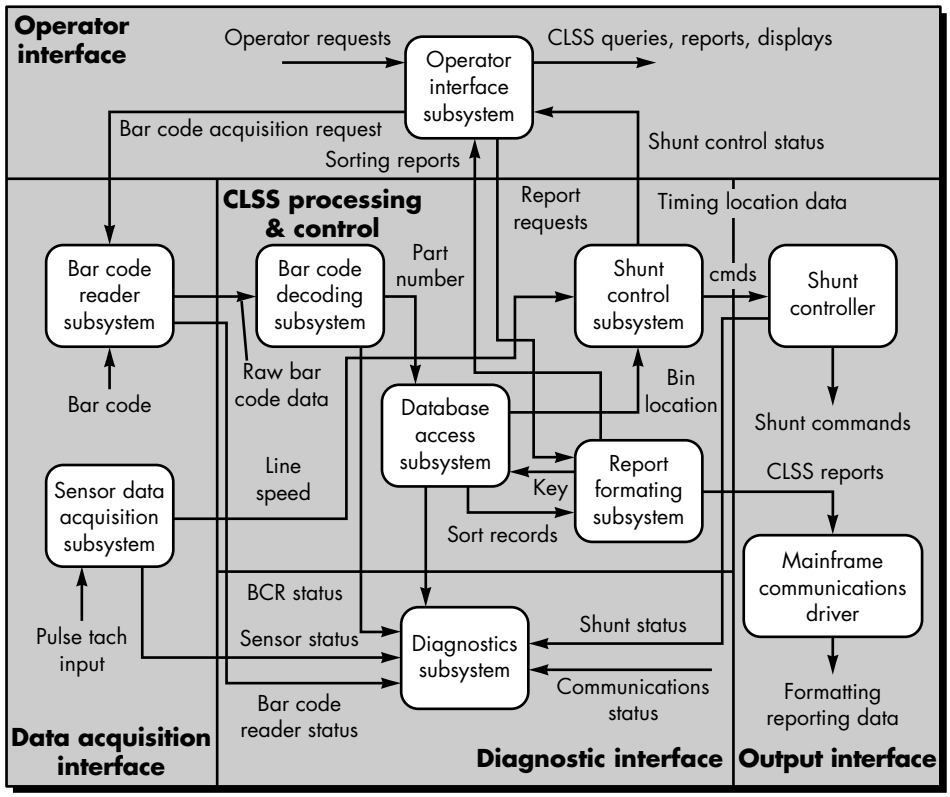


FIGURE 10.7 System flow diagram for CLSS (extended)

Subsystems and the information that flows between them can be specified (bounded) for subsequent engineering work. A narrative description of each subsystem and a definition of all data that flow between subsystems become important elements of the *System Specification*.

10.7 SUMMARY

A high-technology system encompasses a number of elements: software, hardware, people, database, documentation, and procedures. System engineering helps to translate a customer's needs into a model of a system that makes use of one or more of these elements.

System engineering begins by taking a "world view." A business domain or product is analyzed to establish all basic business requirements. Focus is then narrowed to a "domain view," where each of the system elements is analyzed individually. Each element is allocated to one or more engineering components, which are then addressed by the relevant engineering discipline.

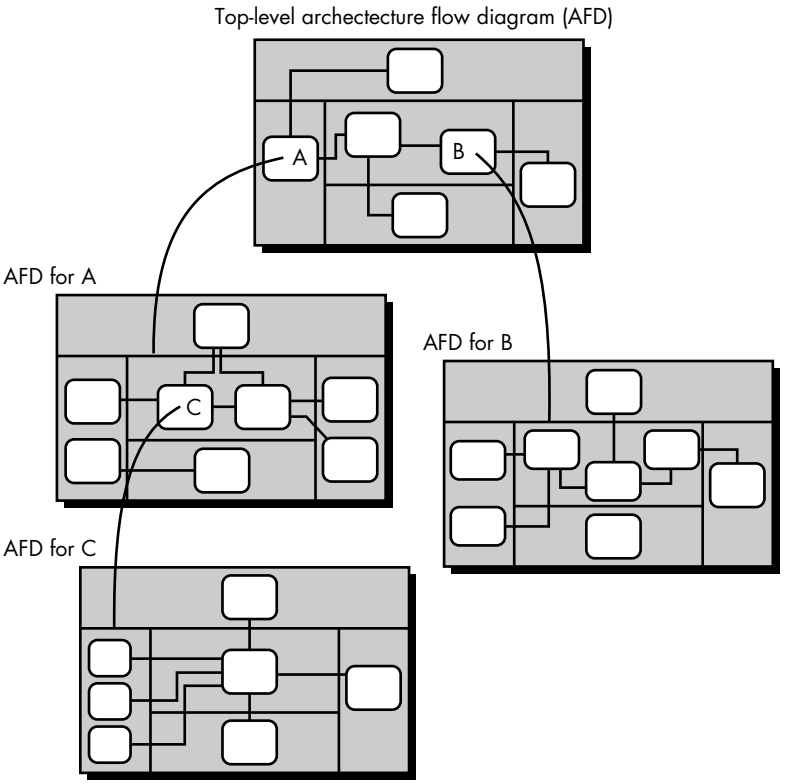


FIGURE 10.8 Building an SFD hierarchy

Business process engineering is a system engineering approach that is used to define architectures that enable a business to use information effectively. The intent of business process engineering is to derive comprehensive data architecture, application architecture, and technology infrastructure that will meet the needs of the business strategy and the objectives and goals of each business area. Business process engineering encompasses information strategy planning (ISP), business area analysis (BAA), and application specific analysis that is actually part of software engineering.

Product engineering is a system engineering approach that begins with system analysis. The system engineer identifies the customer's needs, determines economic and technical feasibility, and allocates function and performance to software, hardware, people, and databases—the key engineering components.

System engineering demands intense communication between the customer and the system engineer. This is achieved through a set of activities that are called requirements engineering—elicitation, analysis and negotiation, specification, modeling, validation, and management.

After requirements have been isolated, a system model is produced and representations of each major subsystem can be developed. The system engineering task culminates with the creation of a *System Specification*—a document that forms the foundation for all engineering work that follows.

REFERENCES

[CRI92] Christel, M.G. and K.C. Kang, "Issues in Requirements Elicitation," Software Engineering Institute, CMU/SEI-92-TR-12 7, September 1992.

[GRA69] Graham, R.M., in *Proceedings 1969 NATO Conference on Software Engineering,* 1969.

[GUT99] Guttman, M., "Architectural Requirements for a Changing Business World," *Research Briefs from Cutter Consortium* (an on-line service), June 1, 1999.

[HAR93] Hares, J.S., *Information Engineering for the Advanced Practitioner,* Wiley, 1993, pp. 12–13.

[HAT87] Hatley, D.J. and I.A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House, 1987.

[MAR90] Martin, J., *Information Engineering: Book II—Planning and Analysis,* Prentice-Hall, 1990.

[MOT92] Motamarri, S., "Systems Modeling and Description," *Software Engineering Notes,* vol. 17, no. 2, April 1992, pp. 57–63.

[SOM97] Somerville, I. and P. Sawyer, *Requirements Engineering,* Wiley, 1997.

[SPE93] Spewak, S., *Enterprise Architecture Planning,* QED Publishing, 1993.

[THA97] Thayer, R.H. and M. Dorfman, *Software Requirements Engineering,* 2nd ed., IEEE Computer Society Press, 1997.

PROBLEMS AND POINTS TO PONDER

10.1. Find as many single-word synonyms for the word *system* as you can. Good luck!

10.2. Build a hierarchical "system of systems" for a system, product, or service with which you are familiar. Your hierarchy should extend down to simple system elements (hardware, software, etc.) along at least one branch of the "tree."

10.3. Select any large system or product with which you are familiar. Define the set of domains that describe the world view of the system or product. Describe the set of elements that make up one or two domains. For one element, identify the technical components that must be engineered.

10.4. Select any large system or product with which you are familiar. State the assumptions, simplifications, limitations, constraints, and preferences that would have to be made to build an effective (and realizable) system model.

10.5. Business process engineering strives to define data and application architecture as well as technology infrastructure. Describe what each of these terms means and provide an example.

10.6. Information strategy planning begins with the definitions of objectives and goals. Provide examples of each from the business domain.

10.7. A system engineer can come from one of three sources: the system developer, the customer, or some outside organization. Discuss the pros and cons that apply to each source. Describe an "ideal" system engineer.

10.8. Your instructor will distribute a high-level description of a computer-based system or product:

a. Develop a set of questions that you should ask as a system engineer.

b. Propose at least two different allocations for the system based on answers to your questions.

c. In class, compare your allocation to those of fellow students.

10.9. Develop a checklist for attributes to be considered when the "feasibility" of a system or product is to be evaluated. Discuss the interplay among attributes and attempt to provide a method for grading each so that a quantitative "feasibility number" may be developed.

10.10. Research the accounting techniques that are used for a detailed cost/benefit analysis of a computer-based system that will require some hardware manufacturing and assembly. Attempt to write a "cookbook" set of guidelines that a technical manager could apply.

10.11. Develop a system context diagram and system flow diagrams for the computer-based system of your choice (or one assigned by your instructor).

10.12. Write a system module narrative that would be contained in system diagram specifications for one or more of the subsystems defined in the SFDs developed for Problem 10.11.

10.13. Research the literature on CASE tools and write a brief paper describing how modeling and simulation tools work. Alternate: Collect literature from two or more CASE vendors that sell modeling and simulation tools and assess the similarities and differences.

10.14. Based on documents provided by your instructor, develop an abbreviated *System Specification* for one of the following computer-based systems:

a. a nonlinear, digital video-editing system

b. a digital scanner for a personal computer

c. an electronic mail system

d. a university registration system

e. an Internet access provider

f. an interactive hotel reservation system

g. a system of local interest

Be sure to create the system models described in Section 10.6.

10.15. Are there characteristics of a system that cannot be established during system engineering activities? Describe the characteristics, if any, and explain why a consideration of them must be delayed until later engineering steps.

10.16. Are there situations in which formal system specification can be abbreviated or eliminated entirely? Explain.

FURTHER READINGS AND INFORMATION SOURCES

Relatively few books have been published on system engineering in recent years. Among those that have appeared are

Blanchard, B.S., *System Engineering Management,* 2nd ed., Wiley, 1997.

Rechtin, E. and M.W. Maier, *The Art of Systems Architecting,* CRC Press, 1996.

Weiss, D., et al., *Software Product-Line Engineering,* Addison-Wesley, 1999.

Books by Armstrong and Sage (*Introduction to Systems Engineering,* Wiley, 1997), Martin (*Systems Engineering Guidebook,* CRC Press, 1996), Wymore (*Model-Based Systems Engineering,* CRC Press, 1993), Lacy (*System Engineering Management,* McGraw-Hill, 1992), Aslaksen and Belcher (*Systems Engineering,* Prentice-Hall, 1992), Athey (*Systematic Systems Approach,* Prentice-Hall, 1982), and Blanchard and Fabrycky (*Systems Engineering and Analysis,* Prentice-Hall, 1981) present the system engineering process (with a distinct engineering emphasis) and provide worthwhile guidance.

In recent years, information engineering texts have been replaced by books that focus on business process engineering. Scheer (*Business Process Engineering: Reference Models for Industrial Enterprises,* Springer-Verlag, 1998) describes business process modeling methods for enterprise-wide information systems. Lozinsky (*Enterprise-wide Software Solutions: Integration Strategies and Practices,* Addison-Wesley, 1998) addresses the use of software packages as a solution that allows a company to migrate from legacy systems to modern business processes. Martin (*Information Engineering,* 3 volumes, Prentice-Hall, 1989, 1990, 1991) presents a comprehensive discussion of information engineering topics. Books by Hares [HAR93], Spewak [SPE93], and Flynn and Fragoso-Diaz (*Information Modeling: An International Perspective,* Prentice-Hall, 1996) also treat the subject in detail.

Davis and Yen (*The Information System Consultant's Handbook: Systems Analysis and Design,* CRC Press, 1998) present encyclopedic coverage of system analysis and design issues in the information systems domain. An excellent IEEE tutorial by Thayer and Dorfman [THA97] discusses the interrelationship between system and software-level requirements analysis issues. A earlier volume by the same authors (*Standards, Guidelines and Examples: System and Software Requirements Engineering,* IEEE Computer Society Press, 1990) presents a comprehensive discussion of standards and guidelines for analysis work.

For those readers actively involved in systems work or interested in a more sophisticated treatment of the topic, Gerald Weinberg's books (*An Introduction to General*

System Thinking, Wiley-Interscience, 1976 and *On the Design of Stable Systems,* Wiley-Interscience, 1979) have become classics and provide an excellent discussion of "general systems thinking" that implicitly leads to a general approach to system analysis and design. More recent books by Weinberg (*General Principles of Systems Design,* Dorset House, 1988 and *Rethinking Systems Analysis and Design,* Dorset House, 1988) continue in the tradition of his earlier work.

A wide variety of information sources on system engineering and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to system engineering, information engineering, business process engineering, and product engineering can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/syseng.mhtml

CHAPTER

11 ANALYSIS CONCEPTS AND PRINCIPLES

KEY CONCEPTS

Software requirements engineering is a process of discovery, refinement, modeling, and specification. The system requirements and role allocated to software—initially established by the system engineer—are refined in detail. Models of the required data, information and control flow, and operational behavior are created. Alternative solutions are analyzed and a complete analysis model is created. Donald Reifer [REI94] describes the software requirement engineering process in the following way:

> Requirements engineering is the systematic use of proven principles, techniques, languages, and tools for the cost effective analysis, documentation, and on-going evolution of user needs and the specification of the external behavior of a system to satisfy those user needs. Notice that like all engineering disciplines, requirements engineering is not conducted in a sporadic, random or otherwise haphazard fashion, but instead is the systematic use of proven approaches.

Both the software engineer and customer take an active role in software requirements engineering—a set of activities that is often referred to as *analysis*. The customer attempts to reformulate a sometimes nebulous system-level description of data, function, and behavior into concrete detail. The developer acts as interrogator, consultant, problem solver, and negotiator.

QUICK LOOK

What is it? The overall role of software in a larger system is identified during system engineering (Chapter 10). However, it's necessary to take a harder look at software's role—to understand the specific requirements that must be achieved to build high-quality software. That's the job of software requirements analysis. To perform the job properly, you should follow a set of underlying concepts and principles.

Who does it? Generally, a software engineer performs requirements analysis. However, for complex business applications, a "system analyst"—trained in the business aspects of the application domain—may perform the task.

Why is it important? If you don't analyze, it's highly likely that you'll build a very elegant software solution that solves the wrong problem. The result is: wasted time and money, personal frustration, and unhappy customers.

What are the steps? Data, functional, and behavioral requirements are identified by eliciting information from the customer. Requirements are refined and analyzed to assess their clarity, completeness, and consistency. A specification incorporating a model of the software is created and then validated by both software engineers and customers/users.

What is the work product? An effective representation of the software must be produced as a

QUICK LOOK

consequence of requirements analysis. Like system requirements, software requirements can be represented using a prototype, a specification or even a symbolic model.

How do I ensure that I've done it right? Software requirements analysis work products must be reviewed for clarity, completeness, and consistency.

Quote:

"This sentence contradicts itself—no actually it doesn't."

Douglas Hofstadter

Requirements analysis and specification may appear to be a relatively simple task, but appearances are deceiving. Communication content is very high. Chances for misinterpretation or misinformation abound. Ambiguity is probable. The dilemma that confronts a software engineer may best be understood by repeating the statement of an anonymous (infamous?) customer: "I know you believe you understood what you think I said, but I am not sure you realize that what you heard is not what I meant."

11.1 REQUIREMENTS ANALYSIS

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design (Figure 11.1). Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behavior), indicate software's interface with other system elements, and establish constraints that software must meet. Requirements analysis allows the software engineer (sometimes called *analyst* in this role) to refine the software allocation and build models of the data, functional, and behavioral domains that will be treated by software. Requirements analysis provides the software designer with a representation of information, function, and behavior that can be translated to data, architectural, interface, and component-level designs. Finally, the requirements specification provides the developer and the customer with the means to assess quality once software is built.

Quote:

"We spend a lot of time—the majority of total project time—not implementing or testing, but trying to decide what to build."

Brian Lawrence

Software requirements analysis may be divided into five areas of effort: (1) problem recognition, (2) evaluation and synthesis, (3) modeling, (4) specification, and (5) review. Initially, the analyst studies the *System Specification* (if one exists) and the *Software Project Plan.* It is important to understand software in a system context and to review the software scope that was used to generate planning estimates. Next, communication for analysis must be established so that problem recognition is ensured. The goal is recognition of the basic problem elements as perceived by the customer/users.

Problem evaluation and solution synthesis is the next major area of effort for analysis. The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behavior in the context of events that affect the system, establish system

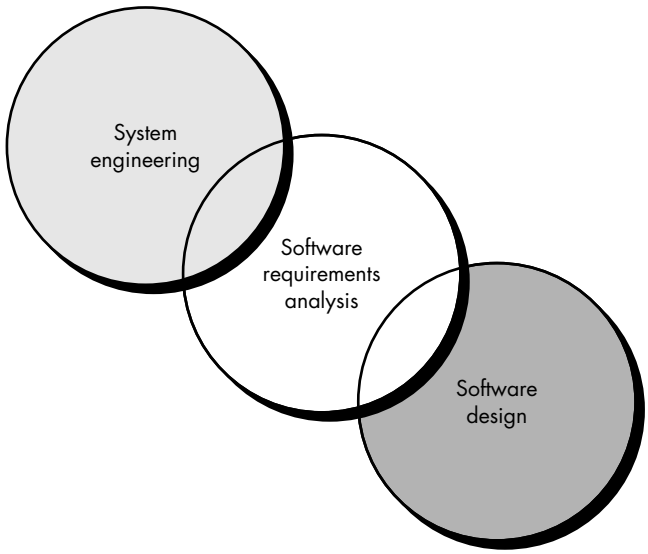


FIGURE 11.1 Analysis as a bridge between system engineering and software design

interface characteristics, and uncover additional design constraints. Each of these tasks serves to describe the problem so that an overall approach or solution may be synthesized.

For example, an inventory control system is required for a major supplier of auto parts. The analyst finds that problems with the current manual system include (1) inability to obtain the status of a component rapidly, (2) two- or three-day turnaround to update a card file, (3) multiple reorders to the same vendor because there is no way to associate vendors with components, and so forth. Once problems have been identified, the analyst determines what information is to be produced by the new system and what data will be provided to the system. For instance, the customer desires a daily report that indicates what parts have been taken from inventory and how many similar parts remain. The customer indicates that inventory clerks will log the identification number of each part as it leaves the inventory area.

Expect to do a bit of design during requirements analysis and a bit of requirements analysis during design.

Upon evaluating current problems and desired information (input and output), the analyst begins to synthesize one or more solutions. To begin, the data objects, processing functions, and behavior of the system are defined in detail. Once this information has been established, basic architectures for implementation are considered. A client/server approach would seem to be appropriate, but does the software to support this architecture fall within the scope outlined in the *Software Plan*? A database management system would seem to be required, but is the user/customer's need for associativity justified? The process of evaluation and synthesis continues until both analyst and customer feel confident that software can be adequately specified for subsequent development steps.

? What should be my primary focus at this stage?

Throughout evaluation and solution synthesis, the analyst's primary focus is on "what," not "how." What data does the system produce and consume, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined and what constraints apply?[1]

During the evaluation and solution synthesis activity, the analyst creates models of the system in an effort to better understand data and control flow, functional processing, operational behavior, and information content. The model serves as a foundation for software design and as the basis for the creation of specifications for the software.

In Chapter 2, we noted that detailed specifications may not be possible at this stage. The customer may be unsure of precisely what is required. The developer may be unsure that a specific approach will properly accomplish function and performance. For these, and many other reasons, an alternative approach to requirements analysis, called *prototyping,* may be conducted. We discuss prototyping later in this chapter.

11.2 REQUIREMENTS ELICITATION FOR SOFTWARE

Before requirements can be analyzed, modeled, or specified they must be gathered through an elicitation process. A customer has a problem that may be amenable to a computer-based solution. A developer responds to the customer's request for help. Communication has begun. But, as we have already noted, the road from communication to understanding is often full of potholes.

11.2.1 Initiating the Process

Quote:

"He who asks a question is a fool for five minutes; he who does not ask a question remains a fool forever."
Chinese Proverb

The most commonly used requirements elicitation technique is to conduct a meeting or interview. The first meeting between a software engineer (the analyst) and the customer can be likened to the awkwardness of a first date between two adolescents. Neither person knows what to say or ask; both are worried that what they do say will be misinterpreted; both are thinking about where it might lead (both likely have radically different expectations here); both want to get the thing over with, but at the same time, both want it to be a success.

Yet, communication must be initiated. Gause and Weinberg [GAU89] suggest that the analyst start by asking *context-free questions.* That is, a set of questions that will lead to a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of the first encounter itself. The first set of context-free questions focuses on the customer, the overall goals, and the benefits. For example, the analyst might ask:

---

1 Davis [DAV93] argues that the terms *what* and *how* are too vague. For an interesting discussion of this issue, the reader should refer to his book.

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables the analyst to gain a better understanding of the problem and the customer to voice his or her perceptions about a solution:

"Plain question and plain answer make the shortest road out of most perplexities."
Mark Twain

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the meeting. Gause and Weinberg [GAU89] call these *meta-questions* and propose the following (abbreviated) list:

If a system or product will serve many users, be absolutely certain that requirements are elicited from a representative cross-section of users. If only one user defines all requirements, acceptance risk is high.

- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions (and others) will help to "break the ice" and initiate the communication that is essential to successful analysis. But a question and answer meeting format is not an approach that has been overwhelmingly successful. In fact, the Q&A session should be used for the first encounter only and then replaced by a meeting format that combines elements of problem solving, negotiation, and specification. An approach to meetings of this type is presented in the next section.

11.2.2 Facilitated Application Specification Techniques

Too often, customers and software engineers have an unconscious "us and them" mind-set. Rather than working as a team to identify and refine requirements, each constituency defines its own "territory" and communicates through a series of memos,

formal position papers, documents, and question and answer sessions. History has shown that this approach doesn't work very well. Misunderstandings abound, important information is omitted, and a successful working relationship is never established.

WebRef

One approach to FAST is called "joint application design" (JAD). A detailed discussion of JAD can be found at **www.bee.net/bluebird/jaddoc.htm**

It is with these problems in mind that a number of independent investigators have developed a team-oriented approach to requirements gathering that is applied during early stages of analysis and specification. Called *facilitated application specification techniques* (FAST), this approach encourages the creation of a joint team of customers and developers who work together to identify the problem, propose elements of the solution, negotiate different approaches and specify a preliminary set of solution requirements [ZAH90]. FAST has been used predominantly by the information systems community, but the technique offers potential for improved communication in applications of all kinds.

Many different approaches to FAST have been proposed.[2] Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

? What makes a FAST meeting different from an ordinary meeting?

- A meeting is conducted at a neutral site and attended by both software engineers and customers.
- Rules for preparation and participation are established.
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A "facilitator" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or an electronic bulletin board, chat room or virtual forum) is used.
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

Quote:

"Facts do not cease to exist because they are ignored."
Aldous Huxley

To better understand the flow of events as they occur in a typical FAST meeting, we present a brief scenario that outlines the sequence of events that lead up to the meeting, occur during the meeting, and follow the meeting.

Initial meetings between the developer and customer (Section 11.2.1) occur and basic questions and answers help to establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customer write a one- or two-page "product request." A meeting place, time, and date for FAST are selected and a facilitator is chosen. Attendees from both the development and customer/user organizations are invited to attend. The product request is distributed to all attendees before the meeting date.

---

2 Two of the more popular approaches to FAST are joint application development (JAD), developed by IBM and the METHOD, developed by Performance Resources, Inc., Falls Church, VA.

KEY POINT

Before the FAST meeting, make a list of objects, services, constraints, and performance criteria.

While reviewing the request in the days before the meeting, each FAST attendee is asked to make a list of *objects* that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of *services* (processes or functions) that manipulate or interact with the objects. Finally, lists of *constraints* (e.g., cost, size, business rules) and *performance criteria* (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

As an example,[3] assume that a FAST team working for a consumer products company has been provided with the following product description:

> Our research indicates that the market for home security systems is growing at a rate of 40 percent per year. We would like to enter this market by building a microprocessor-based home security system that would protect against and/or recognize a variety of undesirable "situations" such as illegal entry, fire, flooding, and others. The product, tentatively called *SafeHome,* will use appropriate sensors to detect each situation, can be programmed by the homeowner, and will automatically telephone a monitoring agency when a situation is detected.

In reality, considerably more information would be provided at this stage. But even with additional information, ambiguity would be present, omissions would likely exist, and errors might occur. For now, the preceding "product description" will suffice.

The FAST team is composed of representatives from marketing, software and hardware engineering, and manufacturing. An outside facilitator is to be used.

KEY POINT

Objects are manipulated by services and must "live" within the constraints and performance defined by the FAST team.

Each person on the FAST team develops the lists described previously. Objects described for *SafeHome* might include smoke detectors, window and door sensors, motion detectors, an alarm, an event (a sensor has been activated), a control panel, a display, telephone numbers, a telephone call, and so on. The list of services might include setting the alarm, monitoring the sensors, dialing the phone, programming the control panel, reading the display (note that services act on objects). In a similar fashion, each FAST attendee will develop lists of constraints (e.g., the system must have a manufactured cost of less than $80, must be user-friendly, must interface directly to a standard phone line) and performance criteria (e.g., a sensor event should be recognized within one second, an event priority scheme should be implemented).

As the FAST meeting begins, the first topic of discussion is the need and justification for the new product—everyone should agree that the product is justified. Once agreement has been established, each participant presents his or her lists for discussion. The lists can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive backed sheets, or written on a wall board. Alternatively, the lists may have been posted on an electronic bulletin board or posed in

---

3 This example (with extensions and variations) will be used to illustrate important software engineering methods in many of the chapters that follow. As an exercise, it would be worthwhile to conduct your own FAST meeting and develop a set of lists for it.

a chat room environment for review prior to the meeting. Ideally, each list entry should be capable of being manipulated separately so that lists can be combined, entries can be deleted and additions can be made. At this stage, critique and debate are strictly prohibited.

ADVICE

Avoid the impulse to shoot down a customer's idea as "too costly" or "impractical." The idea here is to negotiate a list that is acceptable to all. To do this, you must keep an open mind.

After individual lists are presented in one topic area, a combined list is created by the group. The combined list eliminates redundant entries, adds any new ideas that come up during the discussion, but does not delete anything. After combined lists for all topic areas have been created, discussion—coordinated by the facilitator—ensues. The combined list is shortened, lengthened, or reworded to properly reflect the product/system to be developed. The objective is to develop a *consensus list* in each topic area (objects, services, constraints, and performance). The lists are then set aside for later action.

Once the consensus lists have been completed, the team is divided into smaller subteams; each works to develop *mini-specifications* for one or more entries on each of the lists.[4] Each mini-specification is an elaboration of the word or phrase contained on a list. For example, the mini-specification for the *SafeHome* object **control panel** might be

- mounted on wall
- size approximately 9 × 5 inches
- contains standard 12-key pad and special keys
- contains LCD display of the form shown in sketch [not presented here]
- all customer interaction occurs through keys
- used to enable and disable the system
- software provides interaction guidance, echoes, and the like
- connected to all sensors

Each subteam then presents each of its mini-specs to all FAST attendees for discussion. Additions, deletions, and further elaboration are made. In some cases, the development of mini-specs will uncover new objects, services, constraints, or performance requirements that will be added to the original lists. During all discussions, the team may raise an issue that cannot be resolved during the meeting. An issues list is maintained so that these ideas will be acted on later.

After the mini-specs are completed, each FAST attendee makes a list of *validation criteria* for the product/system and presents his or her list to the team. A consensus list of validation criteria is then created. Finally, one or more participants (or outsiders) is assigned the task of writing the complete draft specification using all inputs from the FAST meeting.

Quote:

"The beginning is the most important part of the work."

Plato

4 An alternative approach results in the creation of use-cases. See Section 11.2.4 for details.

FAST is not a panacea for the problems encountered in early requirements elicitation. But the team approach provides the benefits of many points of view, instantaneous discussion and refinement, and is a concrete step toward the development of a specification.

11.2.3 Quality Function Deployment

QFD defines requirements in a way that maximizes customer satisfaction.

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. Originally developed in Japan and first used at the Kobe Shipyard of Mitsubishi Heavy Industries, Ltd., in the early 1970s, QFD "concentrates on maximizing customer satisfaction from the software engineering process [ZUL92]." To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process. QFD identifies three types of requirements [ZUL92]:

Everyone wants to implement lots of exciting requirements, but be careful. That's how "requirements creep" sets in. On the other hand, often the exciting requirements lead to a breakthrough product!

Normal requirements. The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.

Expected requirements. These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction. Examples of expected requirements are: ease of human/machine interaction, overall operational correctness and reliability, and ease of software installation.

Exciting requirements. These features go beyond the customer's expectations and prove to be very satisfying when present. For example, word processing software is requested with standard features. The delivered product contains a number of page layout capabilities that are quite pleasing and unexpected.

In actuality, QFD spans the entire engineering process [AKA90]. However, many QFD concepts are applicable to the requirements elicitation activity. We present an overview of only these concepts (adapted for computer software) in the paragraphs that follow.

The QFD Institute is an excellent source for information:
www.qfdi.org

In meetings with the customer, *function deployment* is used to determine the value of each function that is required for the system. *Information deployment* identifies both the data objects and events that the system must consume and produce. These are tied to the functions. Finally, *task deployment* examines the behavior of the system or product within the context of its environment. *Value analysis* is conducted to determine the relative priority of requirements determined during each of the three deployments.

QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity. These data are then translated into a table of requirements—called the *customer voice table*—that is reviewed with the customer. A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements [BOS91].

11.2.4 Use-Cases

As requirements are gathered as part of informal meetings, FAST, or QFD, the software engineer (analyst) can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use-cases* [JAC92], provide a description of how the system will be used.

A use-case is a scenario that describes how software is to be used in a given situation.

To create a use-case, the analyst must first identify the different types of people (or devices) that use the system or product. These *actors* actually represent roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself.

It is important to note that an actor and a user are not the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore, four actors can be defined: programmer, tester, monitor, and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.

Use-Cases

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors [JAC92] during the first iteration and secondary actors as more is learned about the system. Primary actors interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work.

Use-cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

Once actors have been identified, use-cases can be developed. The use-case describes the manner in which an actor interacts with the system. Jacobson [JAC92] suggests a number of questions that should be answered by the use-case:

- What main tasks or functions are performed by the actor?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?

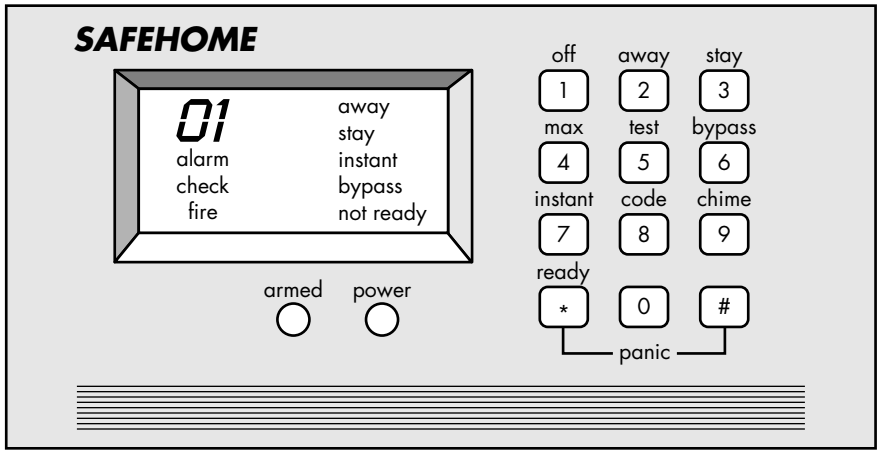


FIGURE 11.2 SafeHome control panel

- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

In general, a use-case is simply a written narrative that describes the role of an actor as interaction with the system occurs.

Recalling basic *SafeHome* requirements (Section 11.2.2), we can define three actors: the homeowner (the user), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors *SafeHome*). For the purposes of this example, we consider only the **homeowner** actor. The homeowner interacts with the product in a number of different ways:

- enters a password to allow all other interactions
- inquires about the status of a security zone
- inquires about the status of a sensor
- presses the panic button in an emergency
- activates/deactivates the security system

A use-case for *system activation* follows:

1. The homeowner observes a prototype of the *SafeHome* control panel (Figure 11.2) to determine if the system is ready for input. If the system is not ready, the homeowner must physically close windows/doors so that the ready indicator is present. [A *not ready* indicator implies that a sensor is open; i.e., that a door or window is open.]
2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for

WebRef

A detailed discussion of use-cases, including examples, guidelines, and templates is presented at **members.aol.com/acockburn/papers/OnUseCases.htm**

additional input. If the password is correct, the control panel awaits further action.

3. The homeowner selects and keys in *stay* or *away* (see Figure 11.2) to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.

Use-cases for other homeowner interactions would be developed in a similar manner. It is important to note that each use-case must be reviewed with care. If some element of the interaction is ambiguous, it is likely that a review of the use-case will indicate a problem.

Each use-case provides an unambiguous scenario of interaction between an actor and the software. It can also be used to specify timing requirements or other constraints for the scenario. For example, in the use-case just noted, requirements indicate that activation occurs 30 seconds after the *stay* or *away* key is hit. This information can be appended to the use-case.

Use-cases describe scenarios that will be perceived differently by different actors. Wyder [WYD96] suggests that quality function deployment can be used to develop a weighted priority value for each use-case. To accomplish this, use-cases are evaluated from the point of view of all actors defined for the system. A priority value is assigned to each use-case (e.g., a value from 1 to 10) by each of the actors.[5] An average priority is then computed, indicating the perceived importance of each of the use-cases. When an iterative process model is used for software engineering, the priorities can influence which system functionality is delivered first.

11.3 ANALYSIS PRINCIPLES

Over the past two decades, a large number of analysis modeling methods have been developed. Investigators have identified analysis problems and their causes and have developed a variety of modeling notations and corresponding sets of heuristics to overcome them. Each analysis method has a unique point of view. However, all analysis methods are related by a set of operational principles:

What are the underlying principles that guide analysis work?

1. The information domain of a problem must be represented and understood.
2. The functions that the software is to perform must be defined.
3. The behavior of the software (as a consequence of external events) must be represented.
4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.

5 Ideally, this evaluation should be performed by individuals from the organization or business function represented by an actor.

5. The analysis process should move from essential information toward implementation detail.

By applying these principles, the analyst approaches a problem systematically. The information domain is examined so that function may be understood more completely. Models are used so that the characteristics of function and behavior can be communicated in a compact fashion. Partitioning is applied to reduce complexity. Essential and implementation views of the software are necessary to accommodate the logical constraints imposed by processing requirements and the physical constraints imposed by other system elements.

In addition to these operational analysis principles, Davis [DAV95a] suggests a set[6] of guiding principles for requirements engineering:

- *Understand the problem before you begin to create the analysis model.* There is a tendency to rush to a solution, even before the problem is understood. This often leads to elegant software that solves the wrong problem!
- *Develop prototypes that enable a user to understand how human/machine interaction will occur.* Since the perception of the quality of software is often based on the perception of the "friendliness" of the interface, prototyping (and the iteration that results) are highly recommended.
- *Record the origin of and the reason for every requirement.* This is the first step in establishing traceability back to the customer.
- *Use multiple views of requirements.* Building data, functional, and behavioral models provide the software engineer with three different views. This reduces the likelihood that something will be missed and increases the likelihood that inconsistency will be recognized.
- *Rank requirements.* Tight deadlines may preclude the implementation of every software requirement. If an incremental process model (Chapter 2) is applied, those requirements to be delivered in the first increment must be identified.
- *Work to eliminate ambiguity.* Because most requirements are described in a natural language, the opportunity for ambiguity abounds. The use of formal technical reviews is one way to uncover and eliminate ambiguity.

Quote:

"A computer will do what you tell it to do, but that may be much different from what you had in mind."

Joseph Weizenbaum

A software engineer who takes these principles to heart is more likely to develop a software specification that will provide an excellent foundation for design.

11.3.1 The Information Domain

All software applications can be collectively called *data processing*. Interestingly, this term contains a key to our understanding of software requirements. Software is built

6 Only a small subset of Davis's requirements engineering principles are noted here. For more information, see [DAV95a].

to process data, to transform data from one form to another; that is, to accept input, manipulate it in some way, and produce output. This fundamental statement of objective is true whether we build batch software for a payroll system or real-time embedded software to control fuel flow to an automobile engine.

The information domain of a problem encompasses data items or objects that contain numbers, text, images, audio, video, or any combination of these.

It is important to note, however, that software also processes events. An event represents some aspect of system control and is really nothing more than Boolean data—it is either on or off, true or false, there or not there. For example, a pressure sensor detects that pressure exceeds a safe value and sends an alarm signal to monitoring software. The alarm signal is an event that controls the behavior of the system. Therefore, data (numbers, text, images, sounds, video, etc.) and control (events) both reside within the information domain of a problem.

The first operational analysis principle requires an examination of the information domain and the creation of a *data model.* The information domain contains three different views of the data and control as each is processed by a computer program: (1) information content and relationships (the data model), (2) information flow, and (3) information structure. To fully understand the information domain, each of these views should be considered.

To begin your understanding of the information domain, the first question to be asked is: "What information does this system produce as output?"

Information content represents the individual data and control objects that constitute some larger collection of information transformed by the software. For example, the data object, **paycheck,** is a composite of a number of important pieces of data: the payee's name, the net amount to be paid, the gross pay, deductions, and so forth. Therefore, the content of **paycheck** is defined by the attributes that are needed to create it. Similarly, the content of a control object called **system status** might be defined by a string of bits. Each bit represents a separate item of information that indicates whether or not a particular device is on- or off-line.

Data and control objects can be related to other data and control objects. For example, the data object paycheck has one or more relationships with the objects timecard, employee, bank, and others. During the analysis of the information domain, these relationships should be defined.

Information flow represents the manner in which data and control change as each moves through a system. Referring to Figure 11.3, input objects are transformed to intermediate information (data and/or control), which is further transformed to output. Along this transformation path (or paths), additional information may be introduced from an existing data store (e.g., a disk file or memory buffer). The transformations applied to the data are functions or subfunctions that a program must perform. Data and control that move between two transformations (functions) define the interface for each function.

Information structure represents the internal organization of various data and control items. Are data or control items to be organized as an *n*-dimensional table or as a hierarchical tree structure? Within the context of the structure, what information is related to other information? Is all information contained within a single structure or

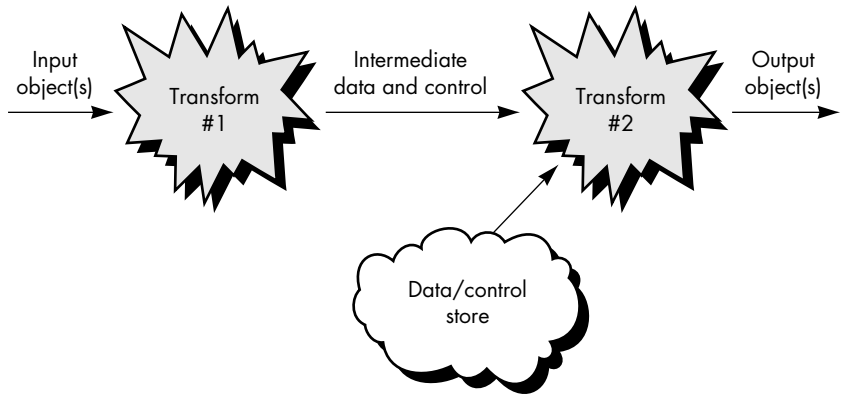


FIGURE 11.3 Information flow and transformation

are distinct structures to be used? How does information in one information structure relate to information in another structure? These questions and others are answered by an assessment of information structure. It should be noted that data structure, a related concept discussed later in this book, refers to the design and implementation of information structure within the software.

11.3.2 Modeling

We create functional models to gain a better understanding of the actual entity to be built. When the entity is a physical thing (a building, a plane, a machine), we can build a model that is identical in form and shape but smaller in scale. However, when the entity to be built is software, our model must take a different form. It must be capable of representing the information that software transforms, the functions (and subfunctions) that enable the transformation to occur, and the behavior of the system as the transformation is taking place.

? What types of models do we create during requirements analysis?

The second and third operational analysis principles require that we build models of function and behavior.

Functional models. Software transforms information, and in order to accomplish this, it must perform at least three generic functions: input, processing, and output. When functional models of an application are created, the software engineer focuses on problem specific functions. The functional model begins with a single context level model (i.e., the name of the software to be built). Over a series of iterations, more and more functional detail is provided, until a thorough delineation of all system functionality is represented.

Behavioral models. Most software responds to events from the outside world. This stimulus/response characteristic forms the basis of the behavioral model. A computer program always exists in some state—an externally observable mode of behavior (e.g., waiting, computing, printing, polling) that is changed only when some event occurs. For example, software will remain

in the wait state until (1) an internal clock indicates that some time interval has passed, (2) an external event (e.g., a mouse movement) causes an interrupt, or (3) an external system signals the software to act in some manner. A behavioral model creates a representation of the states of the software and the events that cause a software to change state.

Models created during requirements analysis serve a number of important roles:

? How do we use the models we create during requirements analysis?

- The model aids the analyst in understanding the information, function, and behavior of a system, thereby making the requirements analysis task easier and more systematic.
- The model becomes the focal point for review and, therefore, the key to a determination of completeness, consistency, and accuracy of the specifications.
- The model becomes the foundation for design, providing the designer with an essential representation of software that can be "mapped" into an implementation context.

The analysis methods that are discussed in Chapters 12 and 21 are actually modeling methods. Although the modeling method that is used is often a matter of personal (or organizational) preference, the modeling activity is fundamental to good analysis work.

11.3.3 Partitioning

KEY POINT

Partitioning is a process that results in the elaboration of data, function, or behavior. It may be performed horizontally or vertically.

Problems are often too large and complex to be understood as a whole. For this reason, we tend to partition (divide) such problems into parts that can be easily understood and establish interfaces between the parts so that overall function can be accomplished. The fourth operational analysis principle suggests that the information, functional, and behavioral domains of software can be partitioned.

In essence, *partitioning* decomposes a problem into its constituent parts. Conceptually, we establish a hierarchical representation of function or information and then partition the uppermost element by (1) exposing increasing detail by moving vertically in the hierarchy or (2) functionally decomposing the problem by moving horizontally in the hierarchy. To illustrate these partitioning approaches, let us reconsider the *SafeHome* security system described in Section 11.2.2. The software allocation for *SafeHome* (derived as a consequence of system engineering and FAST activities) can be stated in the following paragraphs:

SafeHome software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the *SafeHome* control panel shown in Figure 11.2.

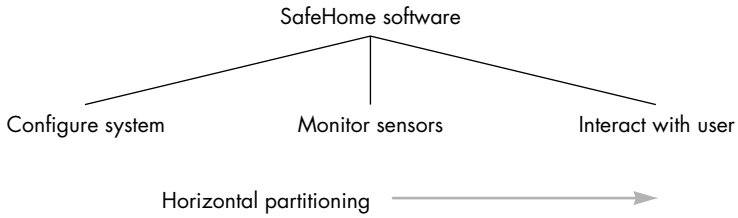


FIGURE 11.4 Horizontal partitioning of SafeHome function

During installation, the *SafeHome* control panel is used to "program" and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.

When a sensor event is recognized, the software invokes an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until telephone connection is obtained.

All interaction with *SafeHome* is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display, displays system status information on the LCD display. Keyboard interaction takes the following form . . .

The requirements for *SafeHome* software may be analyzed by partitioning the information, functional, and behavioral domains of the product. To illustrate, the functional domain of the problem will be partitioned. Figure 11.4 illustrates a *horizontal decomposition* of *SafeHome* software. The problem is partitioned by representing constituent *SafeHome* software functions, moving horizontally in the functional hierarchy. Three major functions are noted on the first level of the hierarchy.

Quote:
"Furious activity is no substitute for understanding."
H. H. Williams

The subfunctions associated with a major *SafeHome* function may be examined by exposing detail vertically in the hierarchy, as illustrated in Figure 11.5. Moving downward along a single path below the function *monitor sensors,* partitioning occurs vertically to show increasing levels of functional detail.

The partitioning approach that we have applied to *SafeHome* functions can also be applied to the information domain and behavioral domain as well. In fact, partitioning of information flow and system behavior (discussed in Chapter 12) will provide additional insight into software requirements. As the problem is partitioned, interfaces between functions are derived. Data and control items that move across an interface should be restricted to inputs required to perform the stated function and outputs that are required by other functions or system elements.

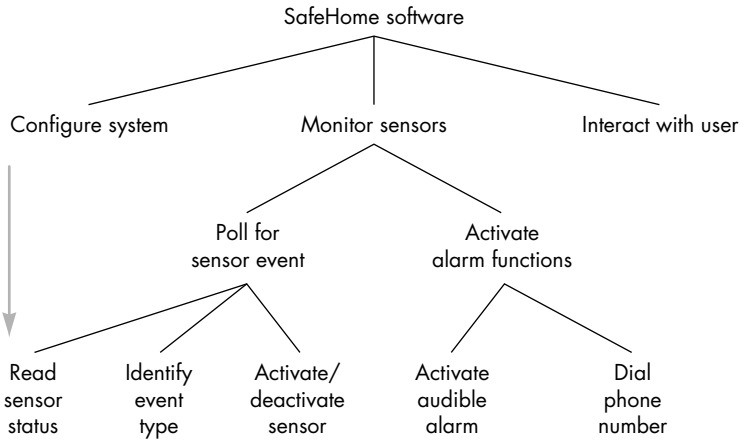


FIGURE 11.5 Vertical partitioning of SafeHome function

11.3.4 Essential and Implementation Views[7]

Avoid the temptation to move directly to the implementation view, assuming that the essence of the problem is obvious. Specifying implementation detail too quickly reduces your options and increases risk.

An *essential view* of software requirements presents the functions to be accomplished and information to be processed without regard to implementation details. For example, the essential view of the *SafeHome* function *read sensor status* does not concern itself with the physical form of the data or the type of sensor that is used. In fact, it could be argued that read status would be a more appropriate name for this function, since it disregards details about the input mechanism altogether. Similarly, an essential data model of the data item **phone number** (implied by the function *dial phone number*) can be represented at this stage without regard to the underlying data structure (if any) used to implement the data item. By focusing attention on the essence of the problem at early stages of requirements engineering, we leave our options open to specify implementation details during later stages of requirements specification and software design.

The *implementation view* of software requirements presents the real world manifestation of processing functions and information structures. In some cases, a physical representation is developed as the first step in software design. However, most computer-based systems are specified in a manner that dictates accommodation of certain implementation details. A *SafeHome* input device is a perimeter sensor (not a watch dog, a human guard, or a booby trap). The sensor detects illegal entry by sensing a break in an electronic circuit. The general characteristics of the sensor should be noted as part of a software requirements specification. The analyst must recognize the constraints imposed by predefined system elements (the sensor) and consider the implementation view of function and information when such a view is appropriate.

---

7 Many people use the terms *logical* and *physical* views to connote the same concept.

We have already noted that software requirements engineering should focus on what the software is to accomplish, rather than on how processing will be implemented. However, the implementation view should not necessarily be interpreted as a representation of how. Rather, an implementation model represents the current mode of operation; that is, the existing or proposed allocation for all system elements. The essential model (of function or data) is generic in the sense that realization of function is not explicitly indicated.

11.4 SOFTWARE PROTOTYPING

Quote:

"Developers may build and test against specifications but users accept or reject against current and actual operational realities."

Bernard Boar

Analysis should be conducted regardless of the software engineering paradigm that is applied. However, the form that analysis takes will vary. In some cases it is possible to apply operational analysis principles and derive a model of software from which a design can be developed. In other situations, requirements elicitation (via FAST, QFD, use-cases, or other "brainstorming" techniques [JOR89]) is conducted, analysis principles are applied, and a model of the software to be built, called a *prototype,* is constructed for customer and developer assessment. Finally, some circumstances require the construction of a prototype at the beginning of analysis, since the model is the only means through which requirements can be effectively derived. The model then evolves into production software.

11.4.1 Selecting the Prototyping Approach

The prototyping paradigm can be either close-ended or open-ended. The close-ended approach is often called *throwaway prototyping.* Using this approach, a prototype serves solely as a rough demonstration of requirements. It is then discarded, and the software is engineered using a different paradigm. An open-ended approach, called *evolutionary prototyping,* uses the prototype as the first part of an analysis activity that will be continued into design and construction. The prototype of the software is the first evolution of the finished system.

? What do I look for to determine whether or not prototyping is a viable approach?

Before a close-ended or open-ended approach can be chosen, it is necessary to determine whether the system to be built is amenable to prototyping. A number of prototyping candidacy factors [BOA84] can be defined: application area, application complexity, customer characteristics, and project characteristics.[8]

In general, any application that creates dynamic visual displays, interacts heavily with a user, or demands algorithms or combinatorial processing that must be developed in an evolutionary fashion is a candidate for prototyping. However, these application areas must be weighed against application complexity. If a candidate application (one that has the characteristics noted) will require the development of tens of thousands of lines of code before any demonstrable function can be performed, it is likely

8 A useful discussion of other candidacy factors—"when to prototype"— can be found in [DAV95b].

FIGURE 11.6 Selecting the appropriate prototyping approach

Question	Throwaway prototype	Evolutionary prototype	Additional preliminary work required
Is the application domain understood?	Yes	Yes	No
Can the problem be modeled?	Yes	Yes	No
Is the customer certain of basic system requirements?	Yes/No	Yes/No	No
Are requirements established and stable?	No	Yes	Yes
Are any requirements ambiguous?	Yes	No	Yes
Are there contradictions in the requirements?	Yes	No	Yes

to be too complex for prototyping.[9] If, however, the complexity can be partitioned, it may still be possible to prototype portions of the software.

Because the customer must interact with the prototype in later steps, it is essential that (1) customer resources be committed to the evaluation and refinement of the prototype and (2) the customer is capable of making requirements decisions in a timely fashion. Finally, the nature of the development project will have a strong bearing on the efficacy of prototyping. Is project management willing and able to work with the prototyping method? Are prototyping tools available? Do developers have experience with prototyping methods? Andriole [AND92] suggests six questions (Figure 11.6) and indicates typical sets of answers and the corresponding suggested prototyping approach.

11.4.2 Prototyping Methods and Tools

For software prototyping to be effective, a prototype must be developed rapidly so that the customer may assess results and recommend changes. To conduct rapid prototyping, three generic classes of methods and tools (e.g., [AND92], [TAN89]) are available:

Fourth generation techniques. Fourth generation techniques (4GT) encompass a broad array of database query and reporting languages, program and application generators, and other very high-level nonprocedural languages. Because 4GT enable the software engineer to generate executable code quickly, they are ideal for rapid prototyping.

Reusable software components. Another approach to rapid prototyping is to assemble, rather than build, the prototype by using a set of existing software components. Melding prototyping and program component reuse will

9 In some cases, extremely complex prototypes can be constructed rapidly by using fourth generation techniques or reusable software components.

work only if a library system is developed so that components that do exist can be cataloged and then retrieved. It should be noted that an existing software product can be used as a prototype for a "new, improved" competitive product. In a way, this is a form of reusability for software prototyping.

Formal specification and prototyping environments. Over the past two decades, a number of formal specification languages and tools have been developed as a replacement for natural language specification techniques. Today, developers of these formal languages are in the process of developing interactive environments that (1) enable an analyst to interactively create language-based specifications of a system or software, (2) invoke automated tools that translate the language-based specifications into executable code, and (3) enable the customer to use the prototype executable code to refine formal requirements.

11.5 SPECIFICATION

There is no doubt that the mode of specification has much to do with the quality of solution. Software engineers who have been forced to work with incomplete, inconsistent, or misleading specifications have experienced the frustration and confusion that invariably results. The quality, timeliness, and completeness of the software suffers as a consequence.

11.5.1 Specification Principles

Specification, regardless of the mode through which we accomplish it, may be viewed as a representation process. Requirements are represented in a manner that ultimately leads to successful software implementation. A number of specification principles, adapted from the work of Balzer and Goodman [BAL86], can be proposed:

In most cases, it is unreasonable to expect that the specification will "cross every t and dot every i." It should, however, capture the essense of what the customer requires.

1. Separate functionality from implementation.
2. Develop a model of the desired behavior of a system that encompasses data and the functional responses of a system to various stimuli from the environment.
3. Establish the context in which software operates by specifying the manner in which other system components interact with software.
4. Define the environment in which the system operates and indicate how "a highly intertwined collection of agents react to stimuli in the environment (changes to objects) produced by those agents" [BAL86].
5. Create a cognitive model rather than a design or implementation model. The cognitive model describes a system as perceived by its user community.
6. Recognize that "the specifications must be tolerant of incompleteness and augmentable." A specification is always a model—an abstraction—of some

real (or envisioned) situation that is normally quite complex. Hence, it will be incomplete and will exist at many levels of detail.

7. Establish the content and structure of a specification in a way that will enable it to be amenable to change.

This list of basic specification principles provides a basis for representing software requirements. However, principles must be translated into realization. In the next section we examine a set of guidelines for creating a specification of requirements.

11.5.2 Representation

We have already seen that software requirements may be specified in a variety of ways. However, if requirements are committed to paper or an electronic presentation medium (and they almost always should be!) a simple set of guidelines is well worth following:

What are a few basic guidelines for representing requirements?

Representation format and content should be relevant to the problem. A general outline for the contents of a *Software Requirements Specification* can be developed. However, the representation forms contained within the specification are likely to vary with the application area. For example, a specification for a manufacturing automation system might use different symbology, diagrams and language than the specification for a programming language compiler.

Information contained within the specification should be nested. Representations should reveal layers of information so that a reader can move to the level of detail required. Paragraph and diagram numbering schemes should indicate the level of detail that is being presented. It is sometimes worthwhile to present the same information at different levels of abstraction to aid in understanding.

Diagrams and other notational forms should be restricted in number and consistent in use. Confusing or inconsistent notation, whether graphical or symbolic, degrades understanding and fosters errors.

Representations should be revisable. The content of a specification will change. Ideally, CASE tools should be available to update all representations that are affected by each change.

Investigators have conducted numerous studies (e.g., [HOL95], [CUR85]) on human factors associated with specification. There appears to be little doubt that symbology and arrangement affect understanding. However, software engineers appear to have individual preferences for specific symbolic and diagrammatic forms. Familiarity often lies at the root of a person's preference, but other more tangible factors such as spatial arrangement, easily recognizable patterns, and degree of formality often dictate an individual's choice.

11.5.3 The Software Requirements Specification

The *Software Requirements Specification* is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements. The National Bureau of Standards, IEEE (Standard No. 830-1984), and the U.S. Department of Defense have all proposed candidate formats for software requirements specifications (as well as other software engineering documentation).

The *Introduction* of the software requirements specification states the goals and objectives of the software, describing it in the context of the computer-based system. Actually, the Introduction may be nothing more than the software scope of the planning document.

Software Requirements Specification

The *Information Description* provides a detailed description of the problem that the software must solve. Information content, flow, and structure are documented. Hardware, software, and human interfaces are described for external system elements and internal software functions.

A description of each function required to solve the problem is presented in the *Functional Description.* A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements. The *Behavioral Description* section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.

When you develop validation criteria, answer the following question: "How would I recognize a successful system if it were dropped on my desk tomorrow?"

Validation Criteria is probably the most important and, ironically, the most often neglected section of the *Software Requirements Specification.* How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints? We neglect this section because completing it demands a thorough understanding of software requirements—something that we often do not have at this stage. Yet, specification of validation criteria acts as an implicit review of all other requirements. It is essential that time and attention be given to this section.

Finally, the specification includes a *Bibliography and Appendix.* The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

In many cases the *Software Requirements Specification* may be accompanied by an executable prototype (which in some cases may replace the specification), a paper prototype or a *Preliminary User's Manual.* The *Preliminary User's Manual* presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

11.6 SPECIFICATION REVIEW

A review of the *Software Requirements Specification* (and/or prototype) is conducted by both the software developer and the customer. Because the specification forms the foundation of the development phase, extreme care should be taken in conducting the review.

Software Requirements Specification Review

The review is first conducted at a macroscopic level; that is, reviewers attempt to ensure that the specification is complete, consistent, and accurate when the overall information, functional, and behavioral domains are considered. However, to fully explore each of these domains, the review becomes more detailed, examining not only broad descriptions but the way in which requirements are worded. For example, when specifications contain "vague terms" (e.g., *some, sometimes, often, usually, ordinarily, most,* or *mostly*), the reviewer should flag the statements for further clarification.

Once the review is complete, the *Software Requirements Specification* is "signed-off" by both the customer and the developer. The specification becomes a "contract" for software development. Requests for changes in requirements after the specification is finalized will not be eliminated. But the customer should note that each after-the-fact change is an extension of software scope and therefore can increase cost and/or protract the schedule.

Even with the best review procedures in place, a number of common specification problems persist. The specification is difficult to "test" in any meaningful way, and therefore inconsistency or omissions may pass unnoticed. During the review, changes to the specification may be recommended. It can be extremely difficult to assess the global impact of a change; that is, how a change in one function affects requirements for other functions. Modern software engineering environments (Chapter 31) incorporate CASE tools that have been developed to help solve these problems.

11.7 SUMMARY

Requirements analysis is the first technical step in the software process. It is at this point that a general statement of software scope is refined into a concrete specification that becomes the foundation for all software engineering activities that follow.

Analysis must focus on the information, functional, and behavioral domains of a problem. To better understand what is required, models are created, the problem is

partitioned, and representations that depict the essence of requirements and, later, implementation detail, are developed.

In many cases, it is not possible to completely specify a problem at an early stage. Prototyping offers an alternative approach that results in an executable model of the software from which requirements can be refined. To properly conduct prototyping special tools and techniques are required.

The *Software Requirements Specification* is developed as a consequence of analysis. Review is essential to ensure that the developer and the customer have the same perception of the system. Unfortunately, even with the best of methods, the problem is that the problem keeps changing.

REFERENCES

[AKA90] Akao, Y., ed., *Quality Function Deployment: Integrating Customer Requirements in Product Design* (translated by G. Mazur), Productivity Press, 1990.

[AND92] Andriole, S., *Rapid Application Prototyping,* QED, 1992.

[BAL86] Balzer, R. and N. Goodman, "Principles of Good Specification and Their Implications for Specification Languages," in *Software Specification Techniques* (Gehani, N. and A. McGetrick, eds.), Addison-Wesley, 1986, pp. 25–39.

[BOA84] Boar, B., *Application Prototyping,* Wiley-Interscience,1984.

[BOS91] Bossert, J.L., *Quality Function Deployment: A Practitioner's Approach,* ASQC Press, 1991.

[CUR85] Curtis, B., *Human Factors in Software Development,* IEEE Computer Society Press, 1985.

[DAV93] Davis, A., *Software Requirements: Objects, Functions and States,* Prentice-Hall, 1993.

[DAV95a] Davis, A., *201 Principles of Software Development,* McGraw-Hill, 1995.

[DAV95b] Davis, A., "Software Prototyping," in *Advances in Computers,* volume 40, Academic Press, 1995.

[GAU89] Gause, D.C. and G.M. Weinberg, *Exploring Requirements: Quality Before Design,* Dorset House, 1989.

[HOL95] Holtzblatt, K. and E. Carmel (eds.), "Requirements Gathering: The Human Factor," special issue of *CACM*, vol. 38, no. 5, May 1995.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[JOR89] Jordan, P.W., et al., "Software Storming: Combining Rapid Prototyping and Knowledge Engineering," *IEEE Computer,* vol. 22, no. 5, May 1989, pp. 39–50.

[REI94] Reifer, D.J., "Requirements Engineering," in *Encyclopedia of Software Engineering* (J.J. Marciniak, ed.), Wiley, 1994, pp. 1043–1054.

[TAN89] Tanik, M.M. and R.T. Yeh (eds.), "Rapid Prototyping in Software Development," special issue of *IEEE Computer,* vol. 22, no. 5, May 1989.

[WYD96] Wyder, T., "Capturing Requirements with Use-Cases," *Software Development,* February 1996, pp. 37–40.

[ZAH90] Zahniser, R.A., "Building Software in Groups," *American Programmer,* vol. 3, nos. 7–8, July-August 1990.
[ZUL92] Zultner, R., "Quality Function Deployment for Software: Satisfying Customers," *American Programmer,* February 1992, pp. 28–41.

PROBLEMS AND POINTS TO PONDER

11.1. Software requirements analysis is unquestionably the most communication-intensive step in the software process. Why does the communication path frequently break down?

11.2. There are frequently severe political repercussions when software requirements analysis (and/or system analysis) begins. For example, workers may feel that job security is threatened by a new automated system. What causes such problems? Can the analysis task be conducted so that politics is minimized?

11.3. Discuss your perceptions of the ideal training and background for a systems analyst.

11.4. Throughout this chapter we refer to the "customer." Describe the "customer" for information systems developers, for builders of computer-based products, for systems builders. Be careful here, there may be more to this problem than you first imagine!

11.5. Develop a facilitated application specification techniques "kit." The kit should include a set of guidelines for conducting a FAST meeting and materials that can be used to facilitate the creation of lists and any other items that might help in defining requirements.

11.6. Your instructor will divide the class into groups of four or six students. Half of the group will play the role of the marketing department and half will take on the role of software engineering. Your job is to define requirements for the *SafeHome* security system described in this chapter. Conduct a FAST meeting using the guidelines presented in this chapter.

11.7. Is it fair to say that a *Preliminary User's Manual* is a form of prototype? Explain your answer.

11.8. Analyze the information domain for *SafeHome.* Represent (using any notation that seems appropriate) information flow in the system, information content, and any information structure that is relevant.

11.9. Partition the functional domain for *SafeHome.* First perform horizontal partitioning; then perform vertical partitioning.

11.10. Create essential and implementation representations of the *SafeHome* system.

11.11. Build a paper prototype (or a real prototype) for *SafeHome.* Be sure to depict owner interaction and overall system function.

11.12. Try to identify software components of *SafeHome* that might be "reusable" in other products or systems. Attempt to categorize these components.

11.13. Develop a written specification for *SafeHome* using the outline provided at the SEPA Web site. (Note: Your instructor will suggest which sections to complete at this time.) Be sure to apply the questions that are described for the specification review.

11.14. How did your requirements differ from others who attempted a solution for *SafeHome?* Who built a "Chevy"—who built a "Cadillac"?

FURTHER READINGS AND INFORMATION SOURCES

Books that address requirements engineering provide a good foundation for the study of basic analysis concepts and principles. Thayer and Dorfman (*Software Requirements Engineering,* 2nd ed., IEEE Computer Society Press, 1997) present a worthwhile anthology on the subject. Graham and Graham (*Requirements Engineering and Rapid Development,* Addison-Wesley, 1998) emphasize rapid development and the use of object-oriented methods in their discussion of requirements engineering, while MacCauley (*Requirements Engineering,* Springer-Verlag, 1996) presents a brief academic treatment of the subject.

In years past, the literature emphasized requirements modeling and specification methods, but today, equal emphasis has been given to effective methods for software requirements elicitation. Wood and Silver (*Joint Application Development,* 2nd ed., Wiley, 1995) have written the definitive treatment of joint application development. Cohen and Cohen (*Quality Function Deployment,* Addison-Wesley, 1995), Terninko (*Step-by-Step QFD: Customer-Driven Product Design,* Saint Lucie Press, 1997), Gause and Weinberg [GAU89], and Zahniser [ZAH90] discuss the mechanics of effective meetings, methods for brainstorming, and elicitation approaches that can be used to clarify results and a variety of other useful issues. Use-cases have become an important part of object-oriented requirements analysis, but they can be used regardless of the implementation technology selected. Rosenburg and Scott (*Use-Case Driven Object Modeling with UML: A Practical Approach,* Addison-Wesley, 1999), Schneider et al. (*Applying Use-Cases: A Practical Guide,* Addison-Wesley, 1998), and Texel and Williams (*Use-Cases Combined With Booch/OMT/UML,* Prentice-Hall, 1997) provide detailed guidance and many useful examples.

Information domain analysis is a fundamental principle of requirements analysis. Books by Mattison (*The Object-Oriented Enterprise*, McGraw-Hill, 1994), Tillman (*A Practical Guide to Logical Data Modeling,* McGraw-Hill, 1993), and Modell (*Data Analysis, Data Modeling and Classification,* McGraw-Hill, 1992) cover various aspects of this important subject.

A recent book by Harrison (*Prototyping and Software Development,* Springer-Verlag, 1999) provides a modern perspective on software prototyping. Two books by Connell and Shafer (*Structured Rapid Prototyping,* Prentice-Hall, 1989) and (*Object-Oriented Rapid Prototyping,* Yourdon Press, 1994) show how this important analysis technique can be used in both conventional and object-oriented environments. Other books by Pomberger et al. (*Object Orientation and Prototyping in Software Engineering,* Prentice-Hall, 1996) and Krief et al. (*Prototyping with Objects,* Prentice-Hall, 1996) examine prototyping from the object-oriented perspective. The *IEEE Proceedings of the International Workshop on Rapid System Prototyping* (published yearly) presents current research in the area.

A wide variety of information sources on requirements analysis and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis concepts and methods can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/reqm.mhtml

CHAPTER

12 ANALYSIS MODELING

KEY CONCEPTS

At a technical level, software engineering begins with a series of modeling tasks that lead to a complete specification of requirements and a comprehensive design representation for the software to be built. The *analysis model,* actually a set of models, is the first technical representation of a system. Over the years many methods have been proposed for analysis modeling. However, two now dominate. The first, *structured analysis,* is a classical modeling method and is described in this chapter. The other approach, *object-oriented analysis,* is considered in detail in Chapter 21. Other commonly used analysis methods are noted in Section 12.8.

Structured analysis is a model building activity. Applying the operational analysis principles discussed in Chapter 11, we create and partition data, functional, and behavioral models that depict the essence of what must built. Structured analysis is not a single method applied consistently by all who use it. Rather, it is an amalgam that evolved over more than 30 years.

In his seminal book on the subject, Tom DeMarco [DEM79] describes structured analysis in this way:

> Looking back over the recognized problems and failings of the analysis phase, I suggest that we need to make the following additions to our set of analysis phase goals:

QUICK LOOK

What is it? The written word is a wonderful vehicle for communication, but it is not necessarily the best way to represent the requirements for computer software. Analysis modeling uses a combination of text and diagrammatic forms to depict requirements for data, function, and behavior in a way that is relatively easy to understand, and more important, straightforward to review for correctness, completeness, and consistency.

Who does it? A software engineer (sometimes called an analyst) builds the model using requirements elicited from the customer.

Why is it important? To validate software requirements, you need to examine them from a number of different points of view. Analysis modeling represents requirements in three "dimensions" thereby increasing the probability that errors will be found, that inconsistency will surface, and that omissions will be uncovered.

What are the steps? Data, functional, and behavioral requirements are modeled using a number of different diagrammatic formats. Data modeling defines data objects, attributes, and relationships. Functional modeling indicates how data are transformed within a system. Behavioral modeling depicts the impact of events. Once preliminary models are created, they are refined and analyzed to assess their clarity, completeness, and consistency. A specification incorporating the

QUICK LOOK

What is the work product? Data object descriptions, entity relationship diagrams, data flow diagrams, state transition diagrams, process specifications, and control specifications are created as part of the analysis modeling activity.

How do I ensure that I've done it right? Analysis modeling work products must be reviewed for correctness, completeness, and consistency.

model is created and then validated by both software engineers and customers/users.

- The products of analysis must be highly maintainable. This applies particularly to the Target Document [software requirements specifications].
- Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out.
- Graphics have to be used whenever possible.
- We have to differentiate between logical [essential] and physical [implementation] considerations . . .

  At the very least, we need . . .
- Something to help us partition our requirements and document that partitioning before specification . . .
- Some means of keeping track of and evaluating interfaces . . .
- New tools to describe logic and policy, something better than narrative text . . .

There is probably no other software engineering method that has generated as much interest, been tried (and often rejected and then tried again) by as many people, provoked as much criticism, and sparked as much controversy. But the method has prospered and has gained a substantial following in the software engineering community.

12.1 A BRIEF HISTORY

Quote:

"The problem is not that there are problems. The problem is expecting otherwise and thinking that having problems is a problem."

Theodore Rubin

Like many important contributions to software engineering, structured analysis was not introduced with a single landmark paper or book. Early work in analysis modeling was begun in the late 1960s and early 1970s, but the first appearance of the structured analysis approach was as an adjunct to another important topic—"structured design." Researchers (e.g., [STE74], [YOU78]) needed a graphical notation for representing data and the processes that transformed it. These processes would ultimately be mapped into a design architecture.

The term *structured analysis,* originally coined by Douglas Ross, was popularized by DeMarco [DEM79]. In his book on the subject, DeMarco introduced and named the key graphical symbols and the models that incorporated them. In the years that followed, variations of the structured analysis approach were suggested by Page-Jones [PAG80], Gane and Sarson [GAN82], and many others. In every instance, the method focused on information systems applications and did not provide an adequate notation to address the control and behavioral aspects of real-time engineering problems.

By the mid-1980s, real-time "extensions" were introduced by Ward and Mellor [WAR85] and later by Hatley and Pirbhai [HAT87]. These extensions resulted in a more robust analysis method that could be applied effectively to engineering problems. Attempts to develop one consistent notation have been suggested [BRU88], and modernized treatments have been published to accommodate the use of CASE tools [YOU89].

12.2 THE ELEMENTS OF THE ANALYSIS MODEL

The analysis model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. To accomplish these objectives, the analysis model derived during structured analysis takes the form illustrated in Figure 12.1.

At the core of the model lies the *data dictionary*—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the the core. The *entity relation diagram* (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data

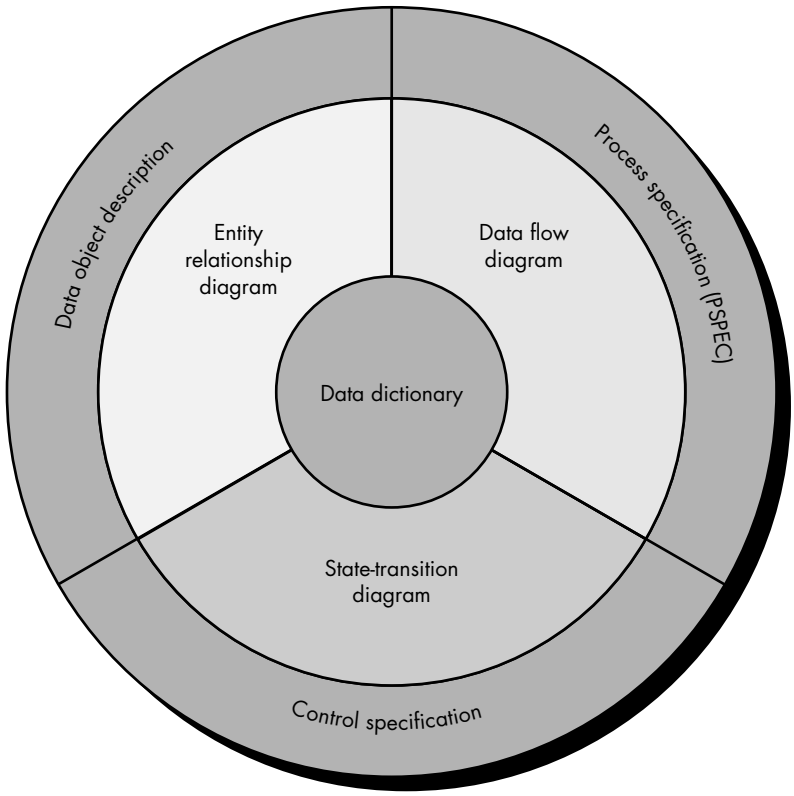


FIGURE 12.1
The structure of the analysis model

modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.

The *data flow diagram* (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and subfunctions) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a *process specification* (PSPEC).

The *state transition diagram* (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called *states*) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control aspects of the software is contained in the *control specification* (CSPEC).

The analysis model encompasses each of the diagrams, specifications, descriptions, and the dictionary noted in Figure 12.1. A more detailed discussion of these elements of the analysis model is presented in the sections that follow.

12.3 DATA MODELING

? What questions does data modeling answer?

Data modeling answers a set of specific questions that are relevant to any data processing application. What are the primary data objects to be processed by the system? What is the composition of each data object and what attributes describe the object? Where do the the objects currently reside? What are the relationships between each object and other objects? What are the relationships between the objects and the processes that transform them?

Quote:

"The power of the ER approach is its ability to describe entities in the real world of the business and the relationships between them."

Martin Modell

To answer these questions, data modeling methods make use of the entity relationship diagram. The ERD, described in detail later in this section, enables a software engineer to identify data objects and their relationships using a graphical notation. In the context of structured analysis, the ERD defines all data that are entered, stored, transformed, and produced within an application.

The entity relationship diagram focuses solely on data (and therefore satisfies the first operational analysis principles), representing a "data network" that exists for a given system. The ERD is especially useful for applications in which data and the relationships that govern data are complex. Unlike the data flow diagram (discussed in Section 12.4 and used to represent how data are transformed), data modeling considers data independent of the processing that transforms the data.

12.3.1 Data Objects, Attributes, and Relationships

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

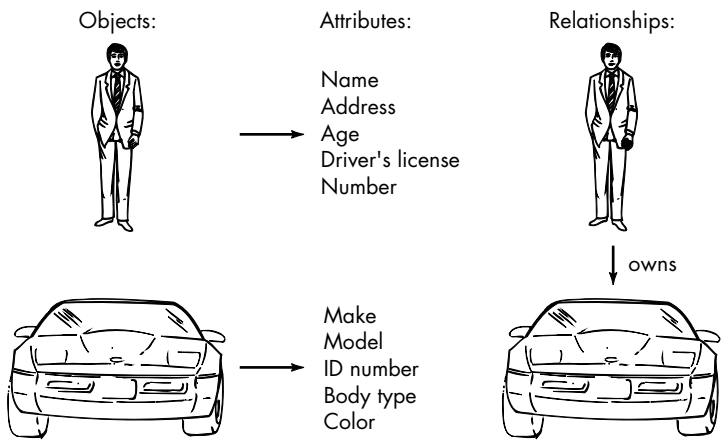


FIGURE 12.2
Data objects, attributes and relationships

Data objects. A *data object* is a representation of almost any composite information that must be understood by software. By *composite information,* we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

KEY POINT
A data object is a representation of any composite information that is processed by computer software.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car (Figure 12.2) can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

WebRef
Useful information on data modeling can be found at **www.datamodel.org**

Data objects (represented in bold) are related to one another. For example, **person** can *own* **car**, where the relationship *own* connotes a specific "connection" between **person** and **car**. The relationships are always defined by the context of the problem that is being analyzed.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.[1] Therefore, the data object can be represented as a table as shown in Figure 12.3. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object **car.**

1 This distinction separates the data object from the *class* or *object* defined as part of the object-oriented paradigm discussed in Part Four of this book.

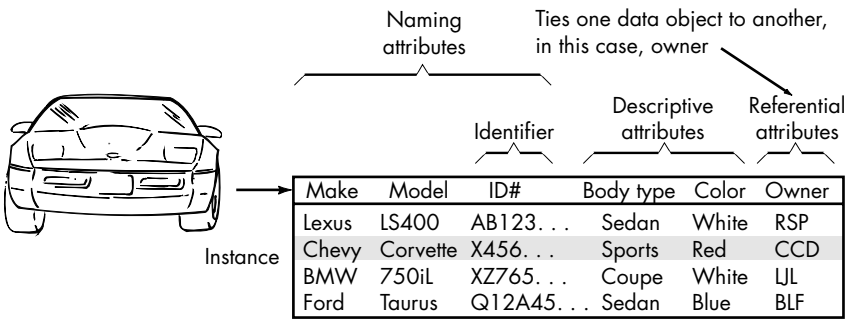


FIGURE 12.3 Tabular representation of data objects

KEY POINT

Attributes name a data object, describe its characteristics, and in some cases, make reference to another object.

Attributes. Attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an *identifier*—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car**, a reasonable identifier might be the ID number.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include ID number, body type and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

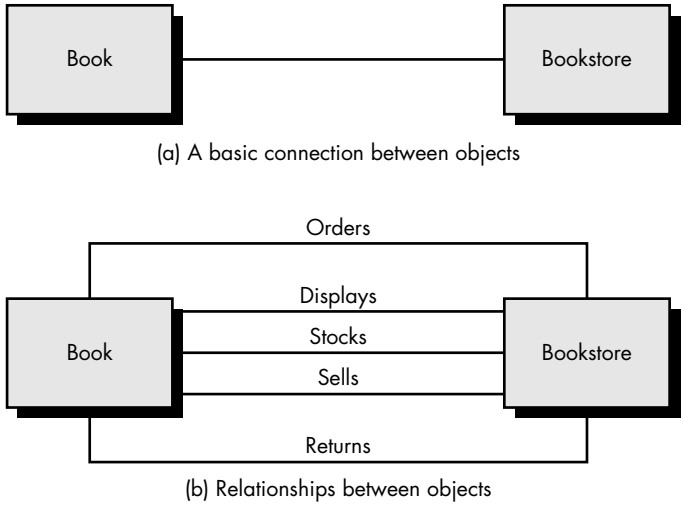
KEY POINT

Relationships indicate the manner in which data objects are "connected" to one another.

Relationships. Data objects are connected to one another in different ways. Consider two data objects, **book** and **bookstore**. These objects can be represented using the simple notation illustrated in Figure 12.4a. A connection is established between **book** and **bookstore** because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

- A bookstore orders books.
- A bookstore displays books.
- A bookstore stocks books.
- A bookstore sells books.
- A bookstore returns books.

FIGURE 12.4
Relationships



The relationships *orders, displays, stocks, sells,* and *returns* define the relevant connections between **book** and **bookstore**. Figure 12.4b illustrates these object/relationship pairs graphically.

It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.[2]

12.3.2 Cardinality and Modality

The elements of data modeling—data objects, attributes, and relationships— provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.

We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: **object X** *relates* to **object Y** does not provide enough information for software engineering purposes. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y.** This leads to a data modeling concept called *cardinality.*

Cardinality. The data model must be capable of representing the number of occurrences objects in a given relationship. Tillmann [TIL93] defines the *cardinality* of an object/relationship pair in the following manner:

2 To avoid ambiguity, the manner in which a relationship is labeled must be considered. For example, if context is not considered for a bidirectional relation, Figure 12.4b could be misinterpreted to mean that books order bookstores. In such cases, rephrasing is necessary.

Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply 'one' or 'many.' For example, a husband can have only one wife (in most cultures), while a parent can have many children. Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as

- One-to-one (l:l)—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'
- One-to-many (l:N)—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.' For example, a mother can have many children, but a child can have only one mother.
- Many-to-many (M:N)—An occurrence of [object] 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.' For example, an uncle can have many nephews, while a nephew can have many uncles.

Cardinality defines "the maximum number of objects that can participate in a relationship" [TIL93]. It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

Modality. The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Figure 12.5 illustrates the relationship, cardinality, and modality between the data objects **customer** and **repair action.**

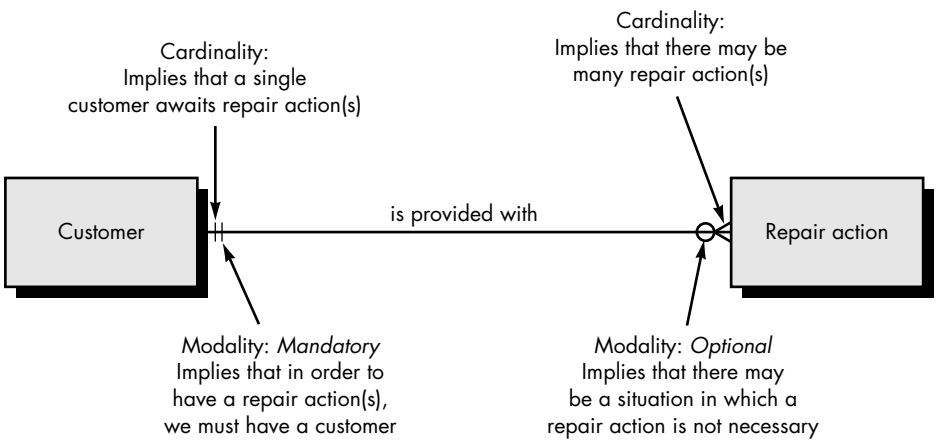


FIGURE 12.5
Cardinality and modality

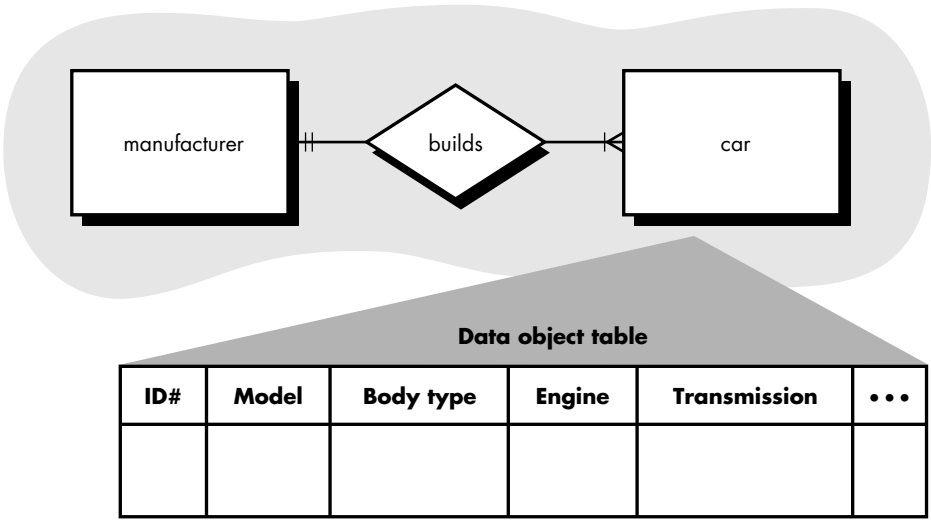


FIGURE 12.6
A simple ERD and data object table (Note: In this ERD the relationship *builds* is indicated by a diamond)

Referring to the figure, a one to many cardinality relationship is established. That is, a single customer can be provided with zero or many repair actions. The symbols on the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar indicates one and the three-pronged fork indicates many. Modality is indicated by the symbols that are further away from the data object rectangles. The second vertical bar on the left indicates that there must be a customer for a repair action to occur. The circle on the right indicates that there may be no repair action required for the type of problem reported by the customer.

KEY POINT
The primary purpose of the ERD is to represent entities (data objects) and their relationships with one another.

12.3.3 Entity/Relationship Diagrams

The object/relationship pair (discussed in Section 12.3.1) is the cornerstone of the data model. These pairs can be represented graphically using the *entity/relationship diagram*. The ERD was originally proposed by Peter Chen [CHE77] for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

Rudimentary ERD notation has already been introduced in Section 12.3. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality (Section 12.3.2).

The relationship between the data objects **car** and **manufacturer** would be represented as shown in Figure 12.6. One manufacturer builds one or many cars. Given

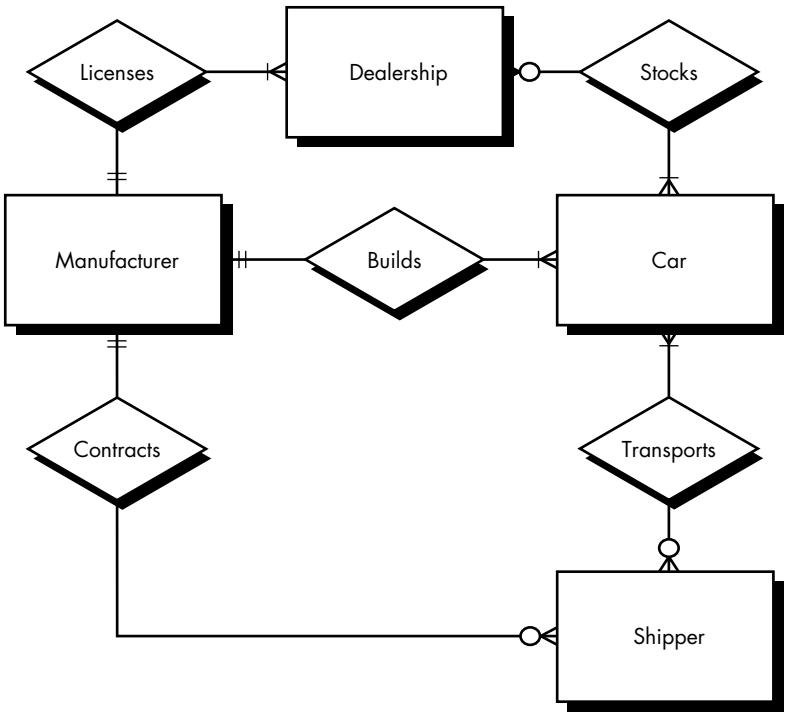


FIGURE 12.7 An expanded ERD

the context implied by the ERD, the specification of the data object **car** (data object table in Figure 12.6) would be radically different from the earlier specification (Figure 12.3). By examining the symbols at the end of the connection line between objects, it can be seen that the modality of both occurrences is mandatory (the vertical lines).

Develop the ERD iteratively by refining both data objects and the relationships that connect them.

Expanding the model, we represent a grossly oversimplified ERD (Figure 12.7) of the distribution element of the automobile business. New data objects, **shipper** and **dealership,** are introduced. In addition, new relationships—*transports, contracts, licenses,* and *stocks*—indicate how the data objects shown in the figure associate with one another. Tables for each of the data objects contained in the ERD would have to be developed according to the rules introduced earlier in this chapter.

In addition to the basic ERD notation introduced in Figures 12.6 and 12.7, the analyst can represent *data object type hierarchies.* In many instances, a data object may actually represent a class or category of information. For example, the data object **car** can be categorized as domestic, European, or Asian. The ERD notation shown in Figure 12.8 represents this categorization in the form of a hierarchy [ROS85].

ERD notation also provides a mechanism that represents the associativity between objects. An *associative data object* is represented as shown in Figure 12.9. In the figure, each of the data objects that model the individual subsystems is associated with the data object **car.**

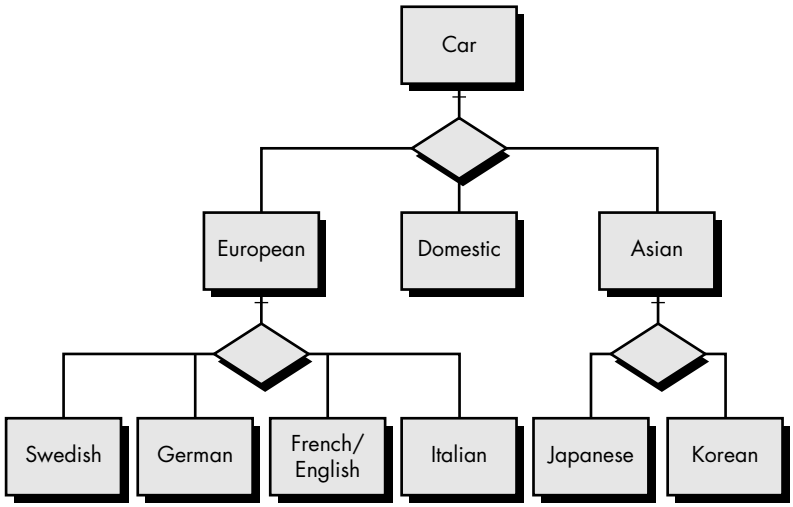


FIGURE 12.8 Data object-type hierarchies

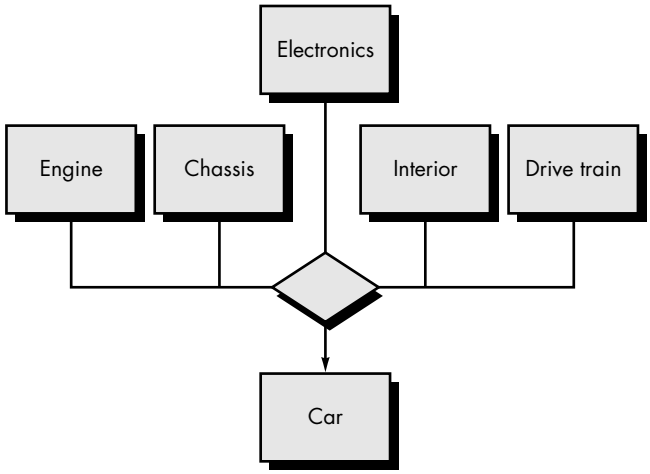


FIGURE 12.9 Associative data objects

Data modeling and the entity relationship diagram provide the analyst with a concise notation for examining data within the context of a software application. In most cases, the data modeling approach is used to create one piece of the analysis model, but it can also be used for database design and to support any other requirements analysis methods.

12.4 FUNCTIONAL MODELING AND INFORMATION FLOW

Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms. Input may be a control

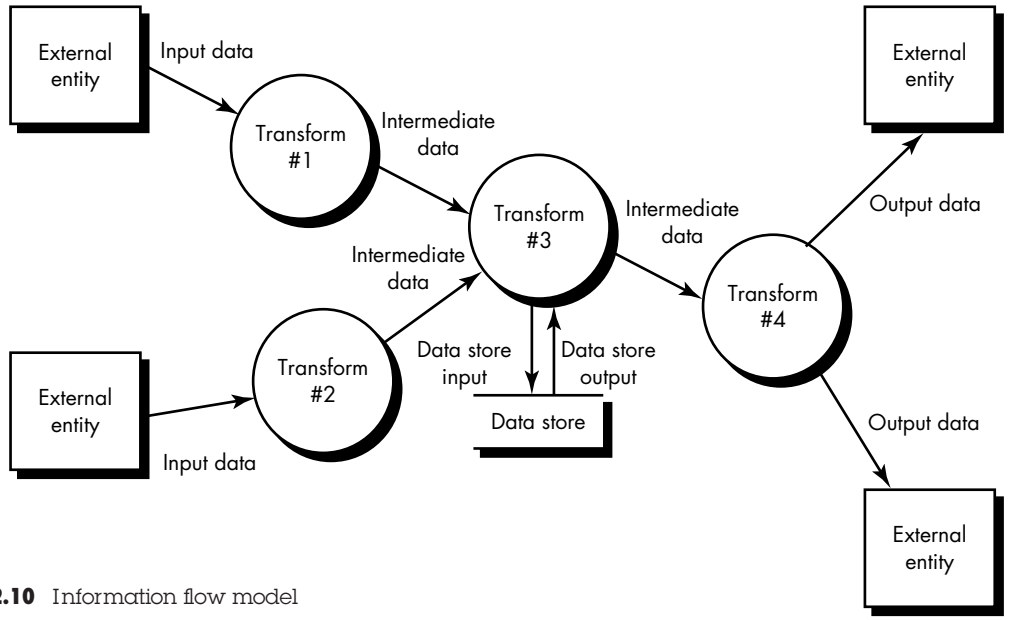


FIGURE 12.10 Information flow model

signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a *flow model* for any computer-based system, regardless of size and complexity.

Structured analysis began as an information flow modeling technique. A computer-based system is represented as an information transform as shown in Figure 12.10. A rectangle is used to represent an *external entity;* that is, a system element (e.g., hardware, a person, another program) or another system that produces information for transformation by the software or receives information produced by the software. A circle (sometimes called a *bubble*) represents a *process* or *transform* that is applied to data (or control) and changes it in some way. An arrow represents one or more *data items* (data objects). All arrows on a data flow diagram should be labeled. The double line represents a data store—stored information that is used by the software. The simplicity of DFD notation is one reason why structured analysis techniques are widely used.

The DFD is not procedural. That is, do not try to represent conditional processing or loops with this diagrammatic form. Simply show the flow of data.

It is important to note that no explicit indication of the sequence of processing or conditional logic is supplied by the diagram. Procedure or sequence may be implicit in the diagram, but explicit logical details are generally delayed until software design. It is important not to confuse a DFD with the flowchart.

12.4.1 Data Flow Diagrams

KEY POINT

The DFD provides a mechanism for information flow modeling and functional modeling.

As information moves through software, it is modified by a series of transformations. A *data flow diagram* is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a *data flow graph* or a *bubble chart,* is illustrated in Figure 12.10.

The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Therefore, the DFD provides a mechanism for functional modeling as well as information flow modeling. In so doing, it satisfies the second operational analysis principle (i.e., creating a functional model) discussed in Chapter 11.

A level 0 DFD, also called a *fundamental system model* or a *context model,* represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 is a subfunction of the overall system depicted in the context model.

ADVICE

Refinement from one DFD level to the next should follow an approximate 1:5 ratio, reducing as the refinement proceeds.

As we noted earlier, each of the bubbles may be refined or layered to depict more detail. Figure 12.11 illustrates this concept. A fundamental model for system *F* indicates the primary input is *A* and ultimate output is *B*. We refine the *F* model into transforms f_1 to f_7. Note that *information flow continuity* must be maintained; that is, input

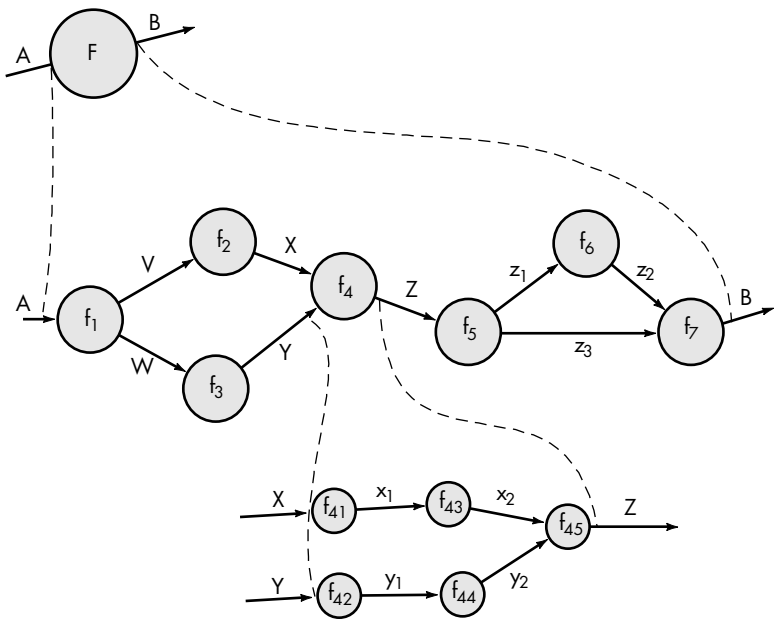


FIGURE 12.11 Information flow refinement

Although information flow continuity must be maintained, recognize that a data item represented at one level may be refined into its constituent parts at the next level.

and output to each refinement must remain the same. This concept, sometimes called *balancing,* is essential for the development of consistent models. Further refinement of f_4 depicts detail in the form of transforms f_{41} to f_{45}. Again, the input (*X, Y*) and output (*Z*) remain unchanged.

The basic notation used to develop a DFD is not in itself sufficient to describe requirements for software. For example, an arrow shown in a DFD represents a data object that is input to or output from a process. A data store represents some organized collection of data. But what is the content of the data implied by the arrow or depicted by the store? If the arrow (or the store) represents a collection of objects, what are they? These questions are answered by applying another component of the basic notation for structured analysis—the *data dictionary.* The use of the data dictionary is discussed later in this chapter.

DFD graphical notation must be augmented with descriptive text. A *process specification* (PSPEC) can be used to specify the processing details implied by a bubble within a DFD. The process specification describes the input to a function, the algorithm that is applied to transform the input, and the output that is produced. In addition, the PSPEC indicates restrictions and limitations imposed on the process (function), performance characteristics that are relevant to the process, and design constraints that may influence the way in which the process will be implemented.

12.4.2 Extensions for Real-Time Systems

Many software applications are time dependent and process as much or more control-oriented information as data. A real-time system must interact with the real world in a time frame dictated by the real world. Aircraft avionics, manufacturing process control, consumer products, and industrial instrumentation are but a few of hundreds of real-time software applications.

To accommodate the analysis of real-time software, a number of extensions to the basic notation for structured analysis have been defined. These extensions, developed by Ward and Mellor [WAR85] and Hatley and Pirbhai [HAT87] and illustrated in the sections that follow, enable the analyst to represent control flow and control processing as well as data flow and processing.

12.4.3 Ward and Mellor Extensions

Ward and Mellor [WAR85] extend basic structured analysis notation to accommodate the following demands imposed by a real-time system:

- Information flow is gathered or produced on a time-continuous basis.
- Control information is passed throughout the system and associated control processing.

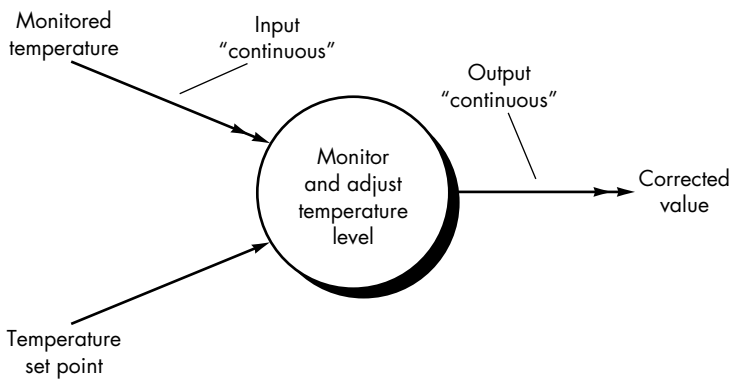


FIGURE 12.12 Time-continuous data flow

- Multiple instances of the same transformation are sometimes encountered in multitasking situations.
- Systems have states and a mechanism causes transition between states.

KEY POINT

To adequately model a real-time system, structured analysis notation must be available for time-continuous data and event processing.

In a significant percentage of real-time applications, the system must monitor time-continuous information generated by some real-world process. For example, a real-time test monitoring system for gas turbine engines might be required to monitor turbine speed, combustor temperature, and a variety of pressure probes on a continuous basis. Conventional data flow notation does not make a distinction between discrete data and time-continuous data. One extension to basic structured analysis notation, shown in Figure 12.12, provides a mechanism for representing *time-continuous data flow.* The double headed arrow is used to represent time-continuous flow while a single headed arrow is used to indicate discrete data flow. In the figure, **monitored temperature** is measured continuously while a single value for **temperature set point** is also provided. The process shown in the figure produces a time-continuous output, **corrected value.**

The distinction between discrete and time-continuous data flow has important implications for both the system engineer and the software designer. During the creation of the system model, a system engineer will be better able to isolate those processes that may be performance critical (it is often likely that the input and output of time-continuous data will be performance sensitive). As the physical or implementation model is created, the designer must establish a mechanism for collection of time-continuous data. Obviously, the digital system collects data in a quasi-continuous fashion using techniques such as high-speed polling. The notation indicates where analog-to-digital hardware will be required and which transforms are likely to demand high-performance software.

In conventional data flow diagrams, control or event flows are not represented explicitly. In fact, the software engineer is cautioned to specifically exclude the

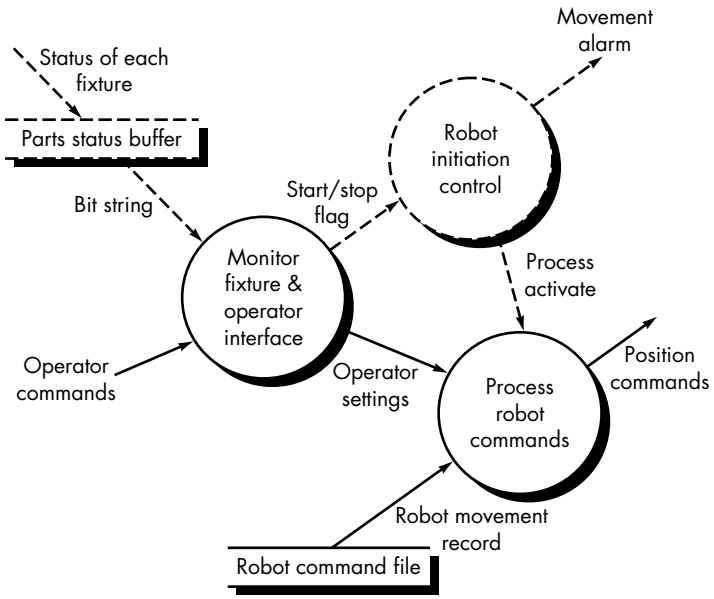


FIGURE 12.13 Data and control flows using Ward and Mellor notation [WAR85]

representation of control flow from the data flow diagram. This exclusion is overly restrictive when real-time applications are considered, and for this reason, a specialized notation for representing event flows and control processing has been developed. Continuing the convention established for data flow diagrams, data flow is represented using a solid arrow. *Control flow,* however, is represented using a dashed or shaded arrow. A process that handles only control flows, called a *control process,* is similarly represented using a dashed bubble.

Quote:

"The environment of a real-time system often contains devices that act as the senses of the system."

Paul Ward and Stephen Mellor

Control flow can be input directly to a conventional process or into a control process. Figure 12.13 illustrates control flow and processing as it would be represented using Ward and Mellor notation. The figure illustrates a top-level view of a data and control flow for a manufacturing cell. As components to be assembled by a robot are placed on fixtures, a status bit is set within a **parts status buffer** (a control store) that indicates the presence or absence of each component. Event information contained within the **parts status buffer** is passed as a bit string to a process, *monitor fixture and operator interface.* The process will read **operator commands** only when the control information, bit string, indicates that all fixtures contain components. An event flag, **start/stop flag,** is sent to *robot initiation control,* a control process that enables further command processing. Other data flows occur as a consequence of the **process activate** event that is sent to *process robot commands.*

In some situations multiple instances of the same control or data transformation process may occur in a real-time system. This can occur in a multitasking environment when tasks are spawned as a result of internal processing or external events. For example, a number of part status buffers may be monitored so that different robots can be signaled at the appropriate time. In addition, each robot may have its own

robot control system. The Ward and Mellor notation used to represent *multiple equivalent instances* simply overlays process (or control process) bubbles to indicate multiplicity.

12.4.4 Hatley and Pirbhai Extensions

The Hatley and Pirbhai [HAT87] extensions to basic structured analysis notation focus less on the creation of additional graphical symbols and more on the representation and specification of the control-oriented aspects of the software. The dashed arrow is once again used to represent control or event flow. Unlike Ward and Mellor, Hatley and Pirbhai suggest that dashed and solid notation be represented separately. Therefore, a *control flow diagram* is defined. The CFD contains the same processes as the DFD, but shows control flow, rather than data flow. Instead of representing control processes directly within the flow model, a notational reference (a solid bar) to a *control specification* (CSPEC) is used. In essence, the solid bar can be viewed as a "window" into an "executive" (the CSPEC) that controls the processes (functions) represented in the DFD based on the event that is passed through the window. The CSPEC, described in detail in Section 12.6.4, is used to indicate (1) how the software behaves when an event or control signal is sensed and (2) which processes are invoked as a consequence of the occurrence of the event. A process specification is used to describe the inner workings of a process represented in a flow diagram.

KEY POINT

The CFD shows how events move through a system. The CSPEC indicates how software behaves as a consequence of events and what processes come into play to manage events.

Using the notation described in Figures 12.12 and 12.13, along with additional information contained in PSPECs and CSPECs, Hatley and Pirbhai create a model of a real-time system. Data flow diagrams are used to represent data and the processes that manipulate it. Control flow diagrams show how events flow among processes and illustrate those external events that cause various processes to be activated. The interrelationship between the process and control models is shown schematically in Figure 12.14. The process model is "connected" to the control model through data conditions. The control model is "connected" to the process model through process activation information contained in the CSPEC.

A *data condition* occurs whenever data input to a process result in control output. This situation is illustrated in Figure 12.15, part of a flow model for an automated monitoring and control system for pressure vessels in an oil refinery. The process *check and convert pressure* implements the algorithm described in the PSPEC pseudocode shown. When the **absolute tank pressure** is greater than an allowable maximum, an **above pressure** event is generated. Note that when Hatley and Pirbhai notation is used, the data flow is shown as part of a DFD, while the control flow is noted separately as part of a control flow diagram. As we noted earlier, the vertical solid bar into which the **above pressure** event flows is a pointer to the CSPEC. Therefore, to determine what happens when this event occurs, we must check the CSPEC.

The control specification (CSPEC) contains a number of important modeling tools. A *process activation table* (described in Section 12.6.4) is used to indicate which

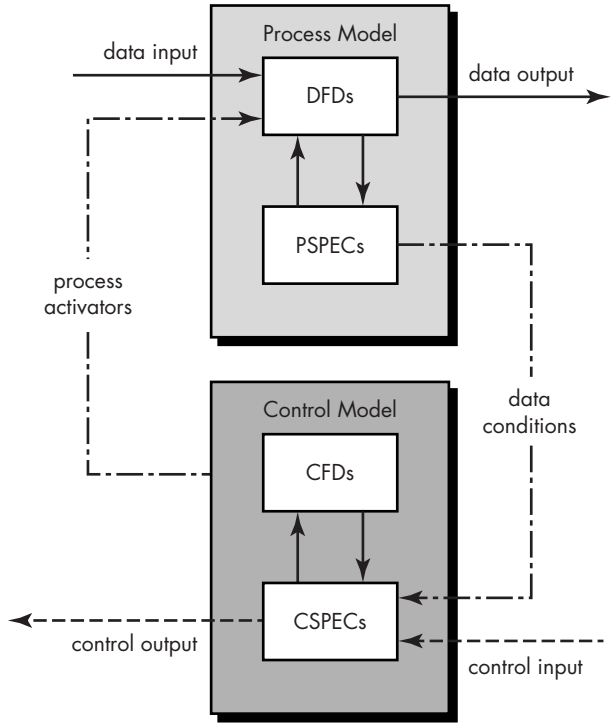


FIGURE 12.14
The relationship between data and control models [HAT87]

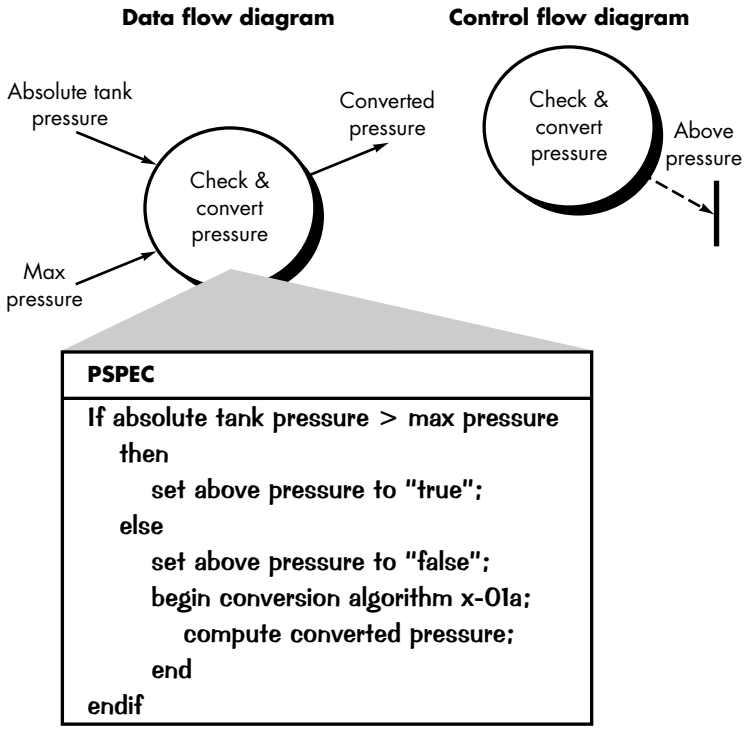


FIGURE 12.15
Data conditions

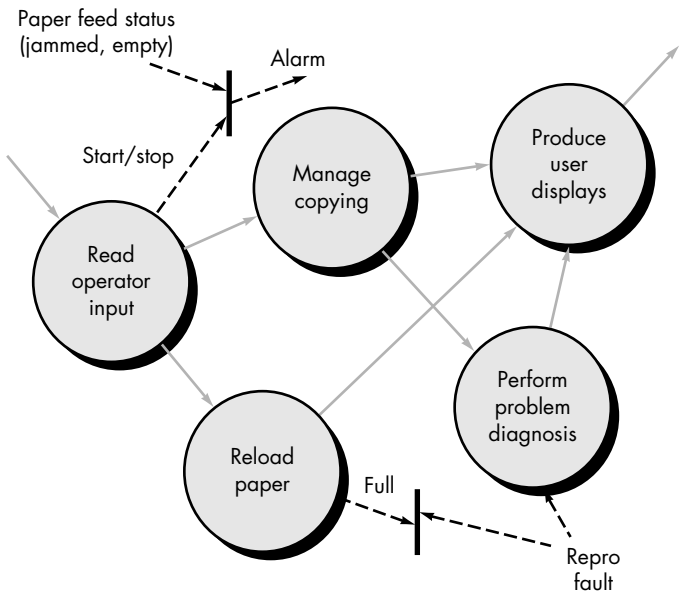


FIGURE 12.16 Level 1 CFD for photocopier software

processes are activated by a given event. For example, a process activation table (PAT) for Figure 12.15 might indicate that the **above pressure** event would cause a process *reduce tank pressure* (not shown) to be invoked. In addition to the PAT, the CSPEC may contain a state transition diagram. The STD is a behavioral model that relies on the definition of a set of system states and is described in the following section.

12.5 BEHAVIORAL MODELING

? How do I model the software's reaction to some external event?

Behavioral modeling is an operational principle for all requirements analysis methods. Yet, only extended versions of structured analysis ([WAR85], [HAT87]) provide a notation for this type of modeling. The state transition diagram represents the behavior of a system by depicting its states and the events that cause the system to change state. In addition, the STD indicates what actions (e.g., process activation) are taken as a consequence of a particular event.

A state is any observable mode of behavior. For example, states for a monitoring and control system for pressure vessels described in Section 12.4.4 might be *monitoring state, alarm state, pressure release state,* and so on. Each of these states represents a mode of behavior of the system. A state transition diagram indicates how the system moves from state to state.

To illustrate the use of the Hatley and Pirbhai control and behavioral extensions, consider software embedded within an office photocopying machine. A simplified representation of the control flow for the photocopier software is shown in Figure 12.16. Data flow arrows have been lightly shaded for illustrative purposes, but in reality they are not shown as part of a control flow diagram.

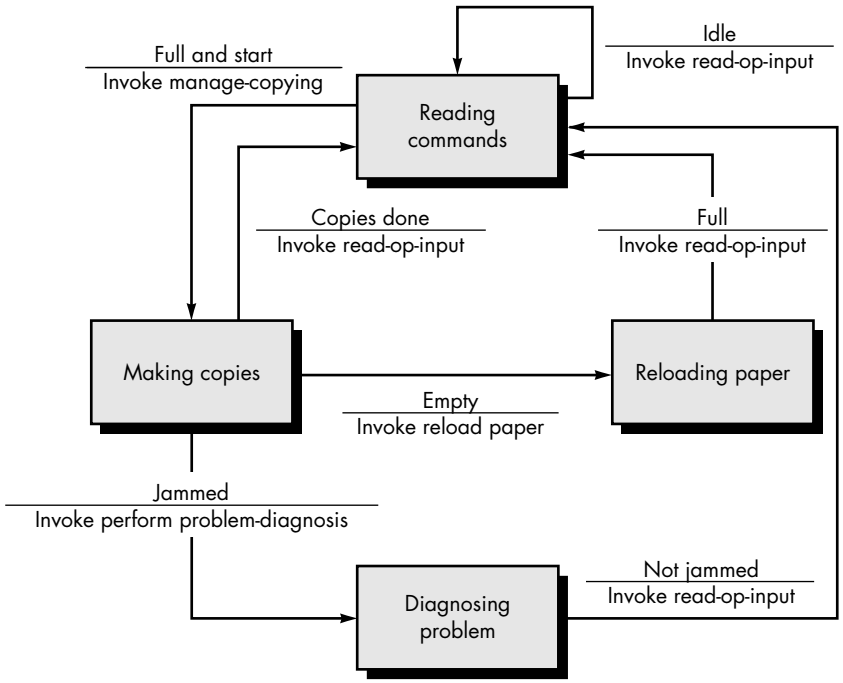


FIGURE 12.17
State transition diagram for photocopier software

Control flows are shown entering and exiting individual processes and the vertical bar representing the CSPEC "window." For example, the **paper feed status** and **start/stop** events flow into the CSPEC bar. This implies that each of these events will cause some process represented in the CFD to be activated. If we were to examine the CSPEC internals, the **start/stop** event would be shown to activate/deactivate the *manage copying* process. Similarly, the **jammed** event (part of **paper feed status**) would activate *perform problem diagnosis.* It should be noted that all vertical bars within the CFD refer to the same CSPEC. An event flow can be input directly into a process as shown with **repro fault.** However, this flow does not activate the process but rather provides control information for the process algorithm.

Quote:

"The only thing missing is a state of confusion."

A reviewer upon puzzling over an extremely complex STD.

A simplified state transition diagram for the photocopier software is shown in Figure 12.17. The rectangles represent system states and the arrows represent transitions between states. Each arrow is labeled with a ruled expression. The top value indicates the event(s) that cause the transition to occur. The bottom value indicates the action that occurs as a consequence of the event. Therefore, when the paper tray is **full** and the **start** button is pressed, the system moves from the *reading commands* state to the *making copies* state. Note that states do not necessarily correspond to processes on a one-to-one basis. For example, the state *making copies* would encompass both the *manage copying* and *produce user displays* processes shown in Figure 12.16.

12.6 THE MECHANICS OF STRUCTURED ANALYSIS

The analysis model allows a reviewer to examine software requirements from three different points of view. Therefore, be certain to use ERDs, DFDs, and STDs when you build the model.

In the previous section, we discussed basic and extended notation for structured analysis. To be used effectively in software requirements analysis, this notation must be combined with a set of heuristics that enable a software engineer to derive a good analysis model. To illustrate the use of these heuristics, an adapted version of the Hatley and Pirbhai [HAT87] extensions to the basic structured analysis notation will be used throughout the remainder of this chapter.

In the sections that follow, we examine each of the steps that should be applied to develop complete and accurate models using structured analysis. Through this discussion, the notation introduced in Section 12.4 will be used, and other notational forms, alluded to earlier, will be presented in some detail.

12.6.1 Creating an Entity/Relationship Diagram

The entity/relationship diagram enables a software engineer to fully specify the data objects that are input and output from a system, the attributes that define the properties of these objects, and their relationships. Like most elements of the analysis model, the ERD is constructed in an iterative manner. The following approach is taken:

What steps are required to build an ERD?

1. During requirements elicitation, customers are asked to list the "things" that the application or business process addresses. These "things" evolve into a list of input and output data objects as well as external entities that produce or consume information.
2. Taking the objects one at a time, the analyst and customer define whether or not a connection (unnamed at this stage) exists between the data object and other objects.
3. Wherever a connection exists, the analyst and the customer create one or more object/relationship pairs.
4. For each object/relationship pair, cardinality and modality are explored.
5. Steps 2 through 4 are continued iteratively until all object/relationships have been defined. It is common to discover omissions as this process continues. New objects and relationships will invariably be added as the number of iterations grows.
6. The attributes of each entity are defined.
7. An entity relationship diagram is formalized and reviewed.
8. Steps 1 through 7 are repeated until data modeling is complete.

To illustrate the use of these basic guidelines, the *SafeHome* security system example, discussed in Chapter 11, will be used. Referring back to the processing narrative

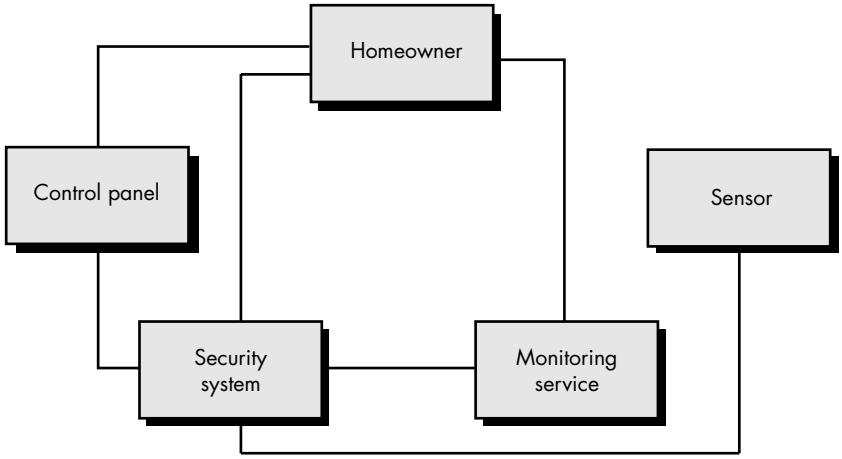


FIGURE 12.18
Establishing connections

for *SafeHome* (Section 11.3.3), the following (partial) list of "things" are relevant to the problem:

- homeowner
- control panel
- sensors
- security system
- monitoring service

Taking these "things" one at a time, connections are explored. To accomplish this, each object is drawn and lines connecting the objects are noted. For example, referring to Figure 12.18, a direct connection exists between **homeowner** and **control panel, security system,** and **monitoring service.** A single connection exists between **sensor** and **security system**, and so forth.

Once all connections have been defined, one or more object/relationship pairs are identified for each connection. For example, the connection between **sensor** and **security system** is determined to have the following object/relationship pairs:

security system *monitors* **sensor**
security system *enables/disables* **sensor**
security system *tests* **sensor**
security system *programs* **sensor**

Each of these object/relationship pairs is analyzed to determine cardinality and modality. For example, considering the object/relationship pair **security system** *monitors* **sensor,** the cardinality between **security system** and **sensor** is one to many. The modality is one occurrence of **security system** (mandatory) and at least one occurrence of **sensor** (mandatory). Using the ERD notation introduced in Section 12.3, the

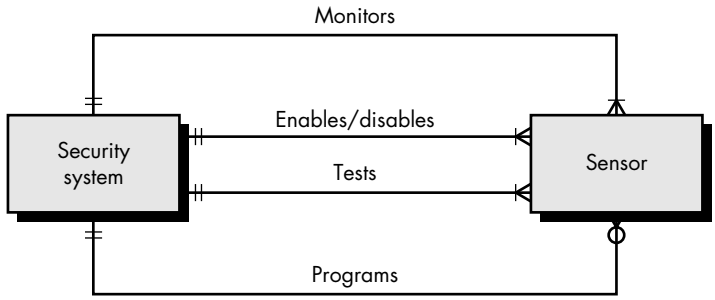


FIGURE 12.19 Developing relationships and cardinality/modality

connecting line between **security system** and **sensor** would be modified as shown in Figure 12.19. Similar analysis would be applied to all other data objects.

Each object is studied to determine its attributes. Since we are considering the software that must support *SafeHome,* the attributes should focus on data that must be stored to enable the system to operate. For example, the **sensor** object might have the following attributes: sensor type, internal identification number, zone location, and alarm level.

12.6.2 Creating a Data Flow Model

The data flow diagram enables the software engineer to develop models of the information domain and functional domain at the same time. As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system, thereby accomplishing the fourth operational analysis principle for function. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

? Are there any useful guidelines for creating DFDs?

A few simple guidelines can aid immeasurably during derivation of a data flow diagram: (1) the level 0 data flow diagram should depict the software/system as a single bubble; (2) primary input and output should be carefully noted; (3) refinement should begin by isolating candidate processes, data objects, and stores to be represented at the next level; (4) all arrows and bubbles should be labeled with meaningful names; (5) information flow continuity must be maintained from level to level, and (6) one bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram. This occurs when the analyst attempts to show too much detail too early or represents procedural aspects of the software in lieu of information flow.

Again considering the *SafeHome* product, a level 0 DFD for the system is shown in Figure 12.20. The primary external entities (boxes) produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object type hierarchies. For example, **user commands and data** encompasses all configuration commands, all activation/deactivation

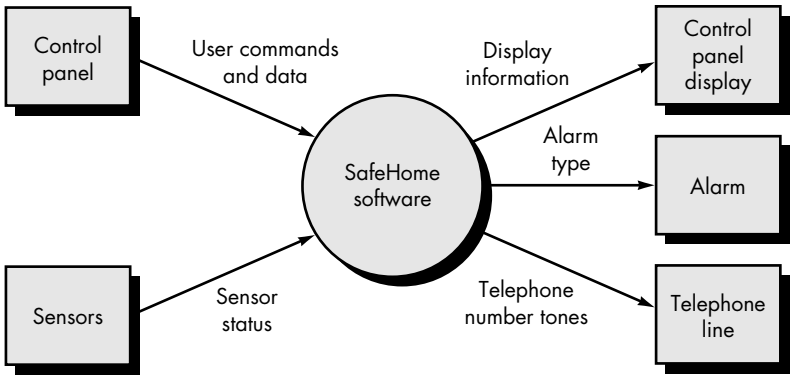


FIGURE 12.20 Context-level DFD for SafeHome

commands, all miscellaneous interactions, and all data that are entered to qualify or expand a command.

The level 0 DFD is now expanded into a level 1 model. But how do we proceed? A simple, yet effective approach is to perform a "grammatical parse" on the processing narrative that describes the context level bubble. That is, we isolate all nouns (and noun phrases) and verbs (and verb phrases) in the *SafeHome* narrative originally presented in Chapter 11. To illustrate, we again reproduce the processing narrative underlining the first occurrence of all nouns and italicizing the first occurrence of all verbs.[3]

The grammatical parse is not foolproof, but it will provide you with an excellent jump start if you're struggling to define data objects and transforms.

SafeHome software *enables* the homeowner to *configure* the security system when it is *installed, monitors* all sensors *connected* to the security system, and *interacts* with the homeowner through a keypad and function keys *contained* in the SafeHome control panel shown in Figure 11.2.

During installation, the SafeHome control panel is *used* to "*program*" and *configure* the system. Each sensor is *assigned* a number and type, a master password is programmed for *arming* and *disarming* the system, and telephone number(s) are *input* for *dialing* when a sensor event occurs.

When a sensor event is *recognized*, the software *invokes* an audible alarm attached to the system. After a delay time that is *specified* by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, *provides* information about the location, *reporting* the nature of the event that has been detected. The telephone number will be *redialed* every 20 seconds until telephone connection is *obtained.*

All interaction with SafeHome is *managed* by a user-interaction subsystem that *reads* input provided through the keypad and function keys, *displays* prompting messages on the LCD display, displays system status information on the LCD display. Keyboard interaction takes the following form . . .

Referring to the "grammatical parse," a pattern begins to emerge. All verbs are *SafeHome* processes; that is, they may ultimately be represented as bubbles in a sub-

3 It should be noted that nouns and verbs that are synonyms or have no direct bearing on the modeling process are omitted.

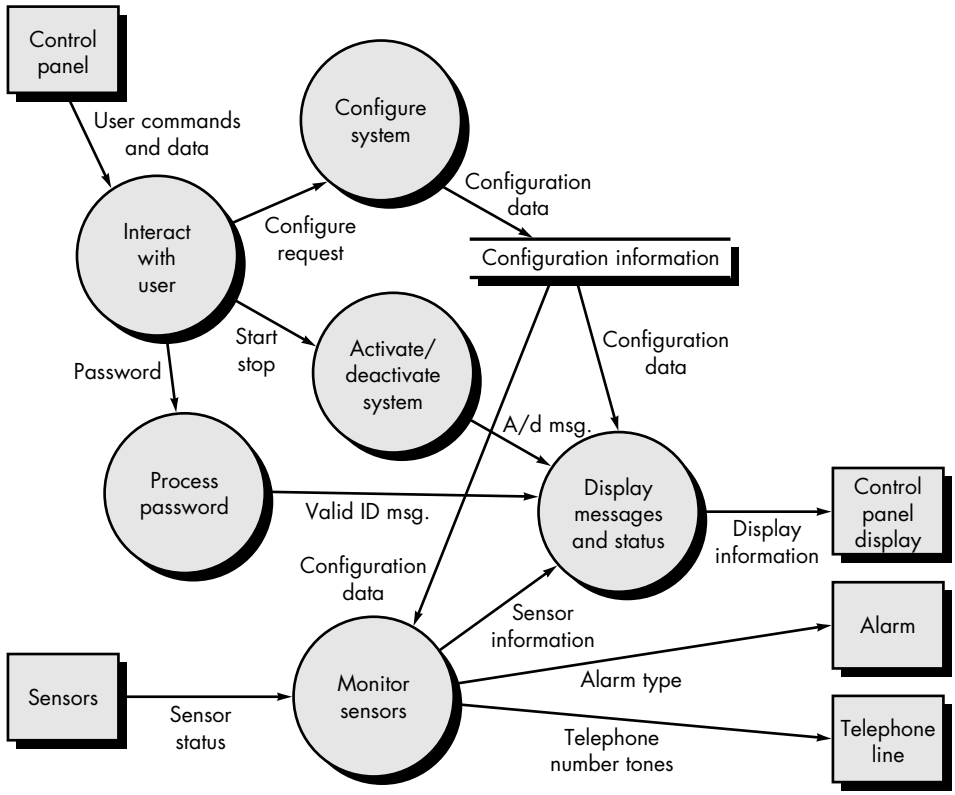


FIGURE 12.21 Level 1 DFD for SafeHome

sequent DFD. All nouns are either external entities (boxes), data or control objects (arrows), or data stores (double lines). Note further that nouns and verbs can be attached to one another (e.g., **sensor** is assigned number and type). Therefore, by performing a grammatical parse on the processing narrative for a bubble at any DFD level, we can generate much useful information about how to proceed with the refinement to the next level. Using this information, a level 1 DFD is shown in Figure 12.21. The context level process shown in Figure 12.20 has been expanded into six processes derived from an examination of the grammatical parse. Similarly, the information flow between processes at level 1 has been derived from the parse.

Be certain that the processing narrative you intend to parse is written at the same level of abstraction throughout.

It should be noted that information flow continuity is maintained between levels 0 and 1. Elaboration of the content of inputs and output at DFD levels 0 and 1 is postponed until Section 12.7.

The processes represented at DFD level 1 can be further refined into lower levels. For example, the process *monitor sensors* can be refined into a level 2 DFD as shown in Figure 12.22. Note once again that information flow continuity has been maintained between levels.

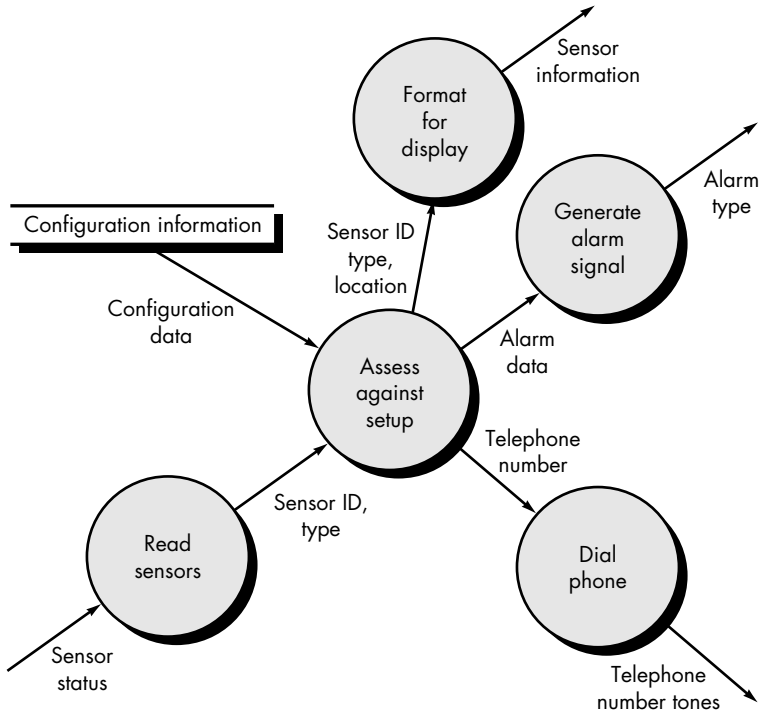


FIGURE 12.22
Level 2 DFD that refines the monitor sensors process

The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. In Chapter 13, we discuss a concept, called *cohesion,* that can be used to assess the simplicity of a given function. For now, we strive to refine DFDs until each bubble is "single-minded."

12.6.3 Creating a Control Flow Model

For many types of data processing applications, the data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. As we have already noted, however, a large class of applications are "driven" by events rather than data; produce control information rather than reports or displays, and process information with heavy concern for time and performance. Such applications require the use of control flow modeling in addition to data flow modeling.

The graphical notation required to create a control flow diagram was presented in Section 12.4.4. To review the approach for creating a CFD, a data flow model is "stripped" of all data flow arrows. Events and control items (dashed arrows) are then added to the diagram and a "window" (a vertical bar) into the control specification is shown. But how are events selected?

We have already noted that an event or control item is implemented as a Boolean value (e.g., true or false, on or off, 1 or 0) or a discrete list of conditions (empty, jammed, full). To select potential candidate events, the following guidelines are suggested:

? How do I select potential events for a CFD, STD, and CSPEC?

- List all sensors that are "read" by the software.
- List all interrupt conditions.
- List all "switches" that are actuated by an operator.
- List all data conditions.
- Recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible CSPEC inputs/outputs.
- Describe the behavior of a system by identifying its states; identify how each state is reached; and define the transitions between states.
- Focus on possible omissions—a very common error in specifying control; for example, ask: "Is there any other way I can get to this state or exit from it?"

A level 1 CFD for *SafeHome* software is illustrated in Figure 12.23. Among the events and control items noted are **sensor event** (i.e., a sensor has been tripped), **blink flag** (a signal to blink the LCD display), and **start/stop switch** (a signal to turn the system on or off). When the event flows into the CSPEC window from the outside world, it implies that the CSPEC will activate one or more of the processes shown in the CFD. When a control item emanates from a process and flows into the CSPEC window, control and activation of some other process or an outside entity is implied.

12.6.4 The Control Specification

The control specification (CSPEC) represents the behavior of the system (at the level from which it has been referenced) in two different ways. The CSPEC contains a state transition diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior. The underlying attributes of the CSPEC were introduced in Section 12.4.4. It is now time to consider an example of this important modeling notation for structured analysis.

Figure 12.24 depicts a state transition diagram for the level 1 control flow model for *SafeHome*. The labeled transition arrows indicate how the system responds to events as it traverses the four states defined at this level. By studying the STD, a software engineer can determine the behavior of the system and, more important, can ascertain whether there are "holes" in the specified behavior. For example, the STD (Figure 12.24) indicates that the only transition from the *reading user input* state occurs when the **start/stop switch** is encountered and a transition to the *monitoring system status* state occurs. Yet, there appears to be no way, other than the occurrence of **sensor event**, that will allow the system to return to *reading user input.* This is an

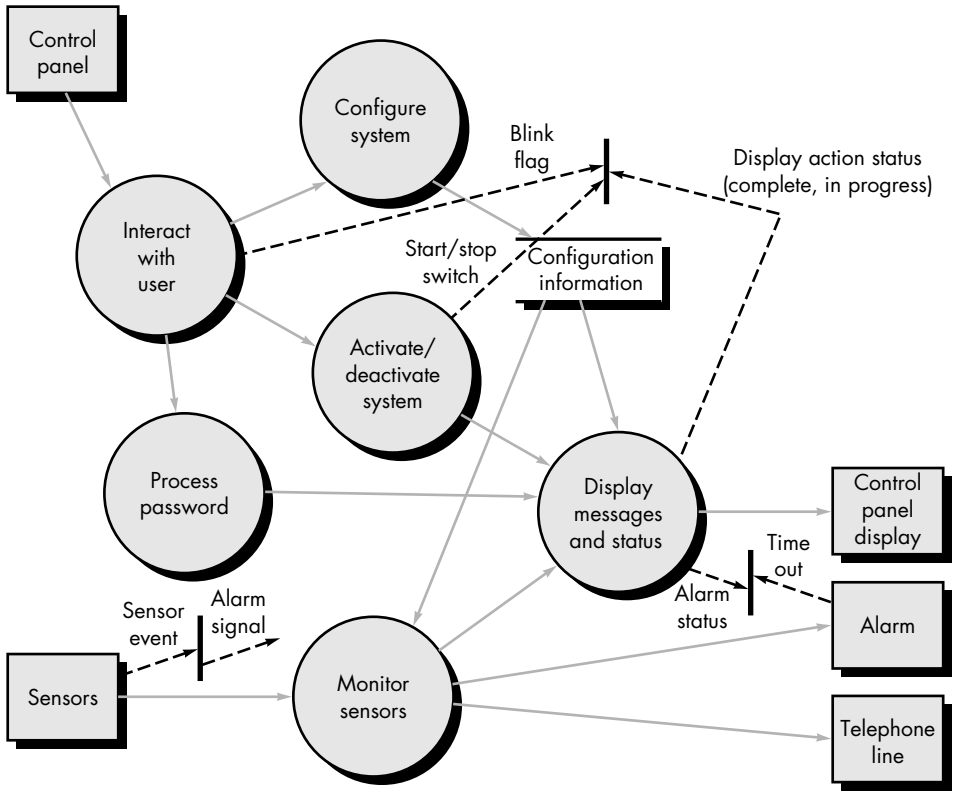
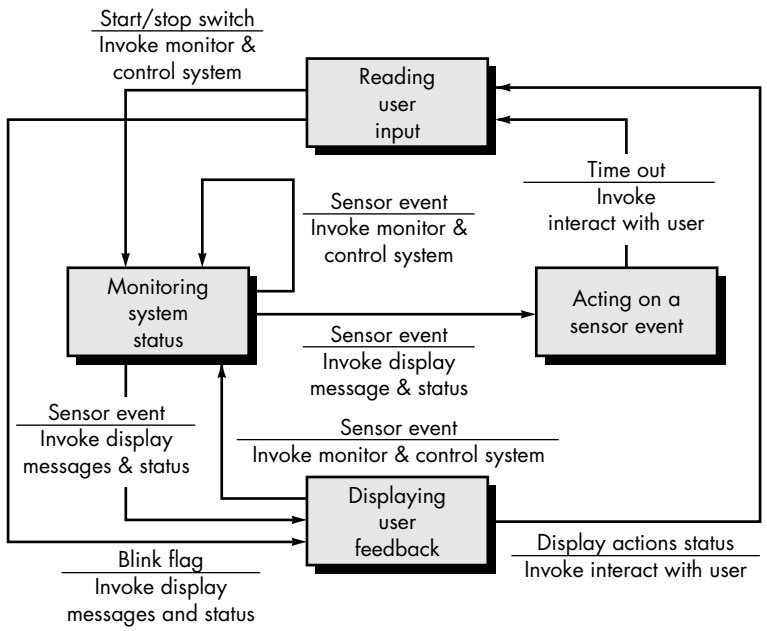


FIGURE 12.23 Level 1 CFD for SafeHome

FIGURE 12.24 State transition diagram for SafeHome



error in specification and would, we hope, be uncovered during review and corrected. Examine the STD to determine whether there are any other anomalies.

A somewhat different mode of behavioral representation is the process activation table. The PAT represents information contained in the STD in the context of processes, not states. That is, the table indicates which processes (bubbles) in the flow model will be invoked when an event occurs. The PAT can be used as a guide for a designer who must build an executive that controls the processes represented at this level. A PAT for the level 1 flow model of *SafeHome* software is shown in Figure 12.25.

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior. The modeling notation that provides this information is discussed in the next section.

12.6.5 The Process Specification

The process specification (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a *program design language* (PDL) description of the process algorithm, mathematical equations, tables, diagrams, or charts. By providing a PSPEC to accompany each bubble in the flow model, the software engineer creates a "mini-spec" that can serve as a first step in the creation of the *Software Requirements Specification* and as a guide for design of the software component that will implement the process.

Input events						
Sensor event	0	0	0	0	1	0
Blink flag	0	0	1	1	0	0
Start/stop switch	0	1	0	0	0	0
Display action status						
Complete	0	0	0	1	0	0
In progress	0	0	1	0	0	0
Time out	0	0	0	0	0	1
Output						
Alarm signal	0	0	0	0	1	0
Process activation						
Monitor and control system	0	1	0	0	1	1
Activate/deactivate system	0	1	0	0	0	0
Display messages and status	1	0	1	1	1	1
Interact with user	1	0	0	1	0	1

FIGURE 12.25 Process activation table for SafeHome

To illustrate the use of the PSPEC, consider the *process password* transform represented in the flow model for *SafeHome* (Figure 12.21). The PSPEC for this function might take the form:

PSPEC: process password

The *process password* transform performs all password validation for the *SafeHome* system. *Process password* receives a four-digit password from the *interact with user* function. The password is first compared to the master password stored within the system. If the master password matches, <valid id message = true> is passed to the *message and status display* function. If the master password does not match, the four digits are compared to a table of secondary passwords (these may be assigned to house guests and/or workers who require entry to the home when the owner is not present). If the password matches an entry within the table, <valid id message = true> is passed to the *message and status display* function. If there is no match, <valid id message = false> is passed to the *message and status display function.*

If additional algorithmic detail is desired at this stage, a program design language representation may also be included as part of the PSPEC. However, many believe that the PDL version should be postponed until component design commences.

12.7 THE DATA DICTIONARY

The analysis model encompasses representations of data objects, function, and control. In each representation data objects and/or control items play a role. Therefore, it is necessary to provide an organized approach for representing the characteristics of each data object and control item. This is accomplished with the data dictionary.

The data dictionary has been proposed as a quasi-formal grammar for describing the content of objects defined during structured analysis. This important modeling notation has been defined in the following manner [YOU89]:

The *data dictionary* is an organized listing of all data elements that are pertinent to the system, with precise, rigorous definitions so that both user and system analyst will have a common understanding of inputs, outputs, components of stores and [even] intermediate calculations.

? How do I represent the content of the data objects I've identified?

Today, the data dictionary is always implemented as part of a CASE "structured analysis and design tool." Although the format of dictionaries varies from tool to tool, most contain the following information:

- *Name*—the primary name of the data or control item, the data store or an external entity.
- *Alias*—other names used for the first entry.
- *Where-used/how-used*—a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.

- *Content description*—a notation for representing content.
- *Supplementary information*—other information about data types, preset values (if known), restrictions or limitations, and so forth.

Once a data object or control item name and its aliases are entered into the data dictionary, consistency in naming can be enforced. That is, if an analysis team member decides to name a newly derived data item **xyz,** but **xyz** is already in the dictionary, the CASE tool supporting the dictionary posts a warning to indicate duplicate names. This improves the consistency of the analysis model and helps to reduce errors.

CASE Tools
Structured Analysis

"Where-used/how-used" information is recorded automatically from the flow models. When a dictionary entry is created, the CASE tool scans DFDs and CFDs to determine which processes use the data or control information and how it is used. Although this may appear unimportant, it is actually one of the most important benefits of the dictionary. During analysis there is an almost continuous stream of changes. For large projects, it is often quite difficult to determine the impact of a change. Many a software engineer has asked, "Where is this data object used? What else will have to change if we modify it? What will the overall impact of the change be?" Because the data dictionary can be treated as a database, the analyst can ask "where used/how used" questions, and get answers to these queries.

The notation used to develop a content description is noted in the following table:

Data Construct	Notation	Meaning
	=	is composed of
Sequence	+	and
Selection	[|]	either-or
Repetition	{ }^{n}	*n* repetitions of
	( )	optional data
	* ... *	delimits comments

The notation enables a software engineer to represent composite data in one of the three fundamental ways that it can be constructed:

1. As a sequence of data items.
2. As a selection from among a set of data items.
3. As a repeated grouping of data items. Each data item entry that is represented as part of a sequence, selection, or repetition may itself be another composite data item that needs further refinement within the dictionary.

To illustrate the use of the data dictionary, we return to the level 2 DFD for the *monitor system* process for *SafeHome,* shown in Figure 12.22. Referring to the figure, the data item **telephone number** is specified as input. But what exactly is a telephone number? It could be a 7-digit local number, a 4-digit extension, or a 25-digit

long distance carrier sequence. The data dictionary provides us with a precise definition of **telephone number** for the DFD in question. In addition it indicates where and how this data item is used and any supplementary information that is relevant to it. The data dictionary entry begins as follows:

```
name:                    telephone number
aliases:                 none
where used/how used:     assess against set-up (output)
                         dial phone (input)
description:
        telephone number = [local number|long distance number]
        local number = prefix + access number
        long distance number = 1 + area code + local number
        area code = [800 | 888 | 561]
        prefix = *a three digit number that never starts with 0 or 1*
        access number = * any four number string *
```

The content description is expanded until all composite data items have been represented as elementary items (items that require no further expansion) or until all composite items are represented in terms that would be well-known and unambiguous to all readers. It is also important to note that a specification of elementary data often restricts a system. For example, the definition of area code indicates that only three area codes (two toll-free and one in South Florida) are valid for this system.

The data dictionary defines information items unambiguously. Although we might assume that the telephone number represented by the DFD in Figure 12.22 could accommodate a 25-digit long distance carrier access number, the data dictionary content description tells us that such numbers are not part of the data that may be used.

For large computer-based systems, the data dictionary grows rapidly in size and complexity. In fact, it is extremely difficult to maintain a dictionary manually. For this reason, CASE tools should be used.

12.8 OTHER CLASSICAL ANALYSIS METHODS

DSSD, JSD, and SADT

Over the years, many other worthwhile software requirements analysis methods have been used throughout the industry. While all follow the operational analysis principles discussed in Chapter 11, each uses a different notation and a unique set of heuristics for deriving the analysis model. An overview of three important analysis methods:

- *Data Structured Systems Development* (DSSD) [WAR81], [ORR81]
- *Jackson System Development* (JSD) [JAC83]
- *Structured Analysis and Design Technique* (SADT) [ROS77], [ROS85]

is presented within the SEPA Web site for those readers interested in a broader view of analysis modeling.

12.9 SUMMARY

Structured analysis, a widely used method of requirements modeling, relies on data modeling and flow modeling to create the basis for a comprehensive analysis model. Using entity-relationship diagrams, the software engineer creates a representation of all data objects that are important for the system. Data and control flow diagrams are used as a basis for representing the transformation of data and control. At the same time, these models are used to create a functional model of the software and to provide a mechanism for partitioning function. A behavioral model is created using the state transition diagram, and data content is developed with a data dictionary. Process and control specifications provide additional elaboration of detail.

The original notation for structured analysis was developed for conventional data processing applications, but extensions have made the method applicable to real-time systems. Structured analysis is supported by an array of CASE tools that assist in the creation of each element of the model and also help to ensure consistency and correctness.

REFERENCES

[BRU88] Bruyn, W. et al., "ESML: An Extended Systems Modeling Language Based on the Data Flow Diagram," *ACM Software Engineering Notes,* vol. 13, no. 1, January 1988, pp. 58–67.

[CHE77] Chen, P., *The Entity-Relationship Approach to Logical Database Design,* QED Information Systems, 1977.

[DEM79] DeMarco, T., *Structured Analysis and System Specification,* Prentice-Hall, 1979.

[GAN82] Gane, T. and C. Sarson, *Structured Systems Analysis,* McDonnell Douglas, 1982.

[HAT87] Hatley, D.J. and I.A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House, 1987.

[JAC83] Jackson, M.A., *System Development,* Prentice-Hall, 1983.

[ORR81] Orr, K.T., *Structured Requirements Definition,* Ken Orr & Associates, Inc., 1981.

[PAG80] Page-Jones, M., *The Practical Guide to Structured Systems Design,* Yourdon Press, 1980.

[ROS77] Ross, D. and K. Schoman, "Structured Analysis for Requirements Definition," *IEEE Trans. Software Engineering,* vol. SE-3, no. 1, January 1977, pp. 6–15.

[ROS85] Ross, D. "Applications and Extensions of SADT," *IEEE Computer,* vol. 18, no. 4, April 1984, pp. 25–35.

[STE74] Stevens, W.P., G.J. Myers, and L.L. Constantine, "Structured Design," *IBM Systems Journal,* vol. 13, no. 2, 1974, pp. 115–139.
[TIL93] Tillmann, G., *A Practical Guide to Logical Data Modeling,* McGraw-Hill, 1993.
[WAR81] Warnier, J.D., *Logical Construction of Systems,* Van Nostrand-Reinhold, 1981.
[WAR85] Ward, P.T. and S.J. Mellor, *Structured Development for Real-Time Systems* (three volumes), Yourdon Press, 1985.
[YOU78] Yourdon, E.N. and Constantine, L.L., *Structured Design,* Yourdon Press, 1978.
[YOU89] Yourdon, E.N., *Modern Structured Analysis,* Prentice-Hall, 1990.

PROBLEMS AND POINTS TO PONDER

12.1. Acquire at least three of the references discussed in Section 12.1 and write a brief paper that outlines how the perception of structured analysis has changed over time. As a concluding section, suggest ways that you think the method will change in the future.

12.2. You have been asked to build one of the following systems:

a. A network-based course registration system for your university.
b. A Web-based order-processing system for a computer store.
c. A simple invoicing system for a small business.
d. Software that replaces a Rolodex and is built into a wireless phone.
e. An automated cookbook that is built into an electric range or microwave.

Select the system that is of interest to you and develop an entity/relationship diagram that describes data objects, relationships, and attributes.

12.3. What is the difference between cardinality and modality?

12.4. Draw a context-level model (level 0 DFD) for one of the five systems that are listed in Problem 12.2. Write a context-level processing narrative for the system.

12.5. Using the context-level DFD developed in Problem 12.4, develop level 1 and level 2 data flow diagrams. Use a "grammatical parse" on the context-level processing narrative to get yourself started. Remember to specify all information flow by labeling all arrows between bubbles. Use meaningful names for each transform.

12.6. Develop a CFDs, CSPECs, PSPECs, and a data dictionary for the system you selected in Problem 12.2. Try to make your model as complete as possible.

12.7. Does the information flow continuity concept mean that, if one flow arrow appears as input at level 0, then one flow arrow must appear as input at subsequent levels? Discuss your answer.

12.8. Using the Ward and Mellor extensions, redraw the flow model contained in Figure 12.16. How will you accommodate the CSPEC that is implied in Figure 12.16? Ward and Mellor do not use this notation.

12.9. Using the Hatley and Pirbhai extensions, redraw the flow model contained in Figure 12.13. How will you accommodate the control process (dashed bubble) that is implied in Figure 12.13? Hatley and Pirbhai do not use this notation.

12.10. Describe an event flow in your own words.

12.11. Develop a complete flow model for the photocopier software discussed in Section 12.5. You may use either the Ward and Mellor or Hatley and Pirbhai method. Be certain to develop a detailed state transition diagram for the system.

12.12. Complete the processing narratives for the analysis model for *SafeHome* software shown in Figure 12.21. Describe the interaction mechanics between the user and the system. Will your additional information change the flow models for *SafeHome* presented in this chapter? If so, how?

12.13. The department of public works for a large city has decided to develop a Web-based *pothole tracking and repair system* (PHTRS). A description follows:

> Citizens can log onto a Web site and report the location and severity of potholes. As potholes are reported they are logged within a "public works department repair system" and are assigned an identifying number, stored by street address, size (on a scale of 1 to 10), location (middle, curb, etc.), district (determined from street address), and repair priority (determined from the size of the pothole). Work order data are associated with each pothole and includes pothole location and size, repair crew identifying number, number of people on crew, equipment assigned, hours applied to repair, hole status (work in progress, repaired, temporary repair, not repaired), amount of filler material used and cost of repair (computed from hours applied, number of people, material and equipment used). Finally, a damage file is created to hold information about reported damage due to the pothole and includes citizen's name, address, phone number, type of damage, dollar amount of damage. PHTRS is an on-line system; all queries are to be made interactively.

Using structured analysis notation, develop a complete analysis model for PHTRS.

12.14. Next generation software for a word-processing system is to be developed. Do a few hours of research on the application area and conduct a FAST meeting (Chapter 11) with your fellow students to develop requirements (your instructor will help you coordinate this). Build a requirements model of the system using structured analysis.

12.15. Software for a video game is to be developed. Proceed as in Problem 12.14.

12.16. Contact four or five vendors that sell CASE tools for structured analysis. Review their literature and write a brief paper that summarizes generic features that seem to distinguish one tool from another.

FURTHER READINGS AND INFORMATION SOURCES

Dozens of books have been published on structured analysis. All cover the subject adequately, but only a few do a truly excellent job. DeMarco's book [DEM79] remains a good introduction to the basic notation. Books by Hoffer et al. (*Modern Systems Analysis and Design,* Addison-Wesley, 2nd ed., 1998), Kendall and Kendall (*Systems Analysis and Design,* 2nd ed., Prentice-Hall, 1998), Davis and Yen (*The Information System Consultant's Handbook: Systems Analysis and Design,* CRC Press, 1998), Modell (*A Professional's Guide to Systems Analysis,* 2nd ed., McGraw-Hill, 1996), Robertson and Robertson (*Complete Systems Analysis,* 2 volumes, Dorset House, 1994), and Page-Jones (*The Practical Guide to Structured Systems Design,* 2nd ed., Prentice-Hall, 1988) are worthwhile references. Yourdon's book on the subject [YOU89] remains among the most comprehensive coverage published to date.

For an engineering emphasis [WAR85] and [HAT87] are the books of preference. However, Edwards (*Real-Time Structured Methods: Systems Analysis,* Wiley, 1993) also covers the analysis of real-time systems in considerable detail, presenting a number of useful examples drawn from actual applications.

Many variations on structured analysis have evolved over the last decade. Cutts (*Structured Systems Analysis and Design Methodology,* Van Nostrand-Reinhold, 1990) and Hares (*SSADM for the Advanced Practitioner,* Wiley, 1990) describe SSADM, a variation on structured analysis that is widely used in the United Kingdom and Europe.

Flynn et al. (*Information Modeling: An International Perspective,* Prentice-Hall, 1996), Reingruber and Gregory (*Data Modeling Handbook,* Wiley, 1995) and Tillman [TIL93] present detailed tutorials for creating industry-quality data models. Kim and Salvatore ("Comparing Data Modeling Formalisms," *Communications of the ACM,* June 1995) have written an excellent comparison of data modeling methods. An interesting book by Hay (*Data Modeling Patterns,* Dorset House, 1995) presents typical data model "patterns" that are encountered in many different businesses. A detailed treatment of behavioral modeling can be found in Kowal (*Behavior Models: Specifying User's Expectations,* Prentice-Hall, 1992).

A wide variety of information sources on structured analysis and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to analysis concepts and methods can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/reqm-analysis.mhtml

CHAPTER

13 DESIGN CONCEPTS AND PRINCIPLES

KEY CONCEPTS

The designer's goal is to produce a model or representation of an entity that will later be built. The process by which the design model is developed is described by Belady [BEL81]:

> [T]here are two major phases to any design process: diversification and convergence. Diversification is the *acquisition* of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind. During convergence, the designer chooses and combines appropriate elements from this repertoire to meet the design objectives, as stated in the requirements document and as agreed to by the customer. The second phase is the gradual *elimination* of all but one particular configuration of components, and thus the creation of the final product.

Diversification and convergence combine intuition and judgment based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

Software design, like engineering design approaches in other disciplines, changes continually as new methods, better analysis, and broader understanding

QUICK LOOK

What is it? Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components. The concepts and principles discussed in this chapter apply to all four.

Who does it? Software engineers design computer-based systems, but the skills required at each level of design work are different. At the data and architectural level, design focuses on patterns as they apply to the application to be built. At the interface level, human ergonomics often dictate our design approach. At the component level, a "programming approach" leads us to effective data and procedural designs.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You'd risk confusion, errors, a floor plan that didn't make sense, windows and doors in the wrong place . . . a mess. Computer software is considerably more complex than a house; hence, we need a blueprint—the design.

What are the steps? Design begins with the requirements model. We work to transform this model into four levels of design detail: the data structure,