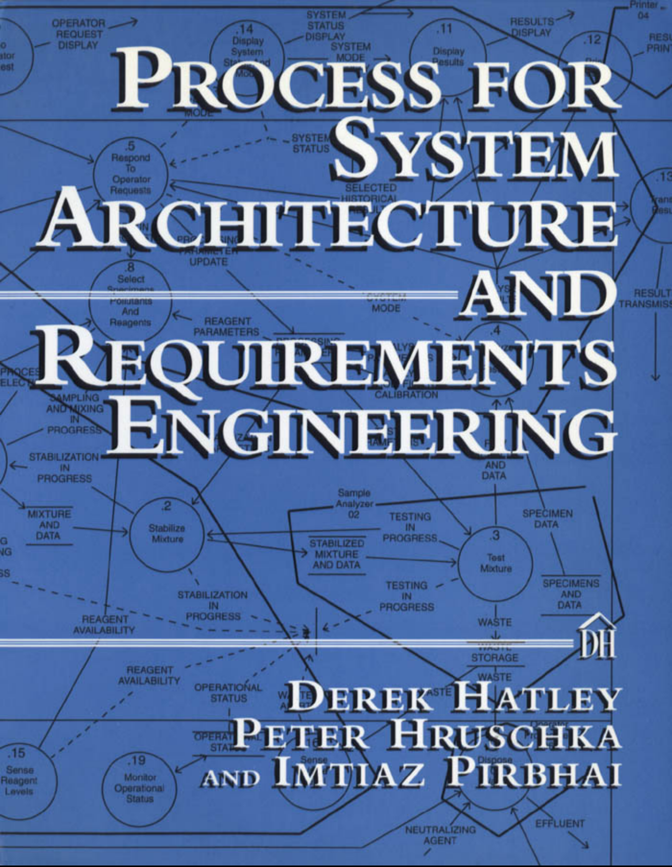
Process for System Architecture and Requirements Engineering

Derek Hatley
Peter Hruschka
and Imtiaz Pirbhai



Process for System Architecture and Requirements Engineering

Process for System Architecture and Requirements Engineering

Derek Hatley   Peter Hruschka   Imtiaz Pirbhai

Dorset House Publishing
353 West 12th Street
New York, NY 10014

Library of Congress Cataloging-in-Publication Data

Hatley, Derek J., 1934-
Process for system architecture and requirements engineering / Derek Hatley, Peter Hruschka, Imtiaz Pirbhai.
p. cm.
ISBN 0-932633-41-2 (pbk.)
1. System design. 2. System analysis. I. Hruschka, Peter, 1951- II. Pirbhai, Imtiaz A., 1953- III. Title.

QA76.9.S88 H3735 2000
005.1'2--dc21

00-060997

Cover Design: David W. McClintock
Cover Graphic: Detail from Figure 11.2
Cover Author Photographs: Hatley photo: O'Connor-Rice Studios; Hruschka photo: James Robertson; Pirbhai photo: Stewart Auer.

Distributed in the English language in Singapore, the Philippines, and Southeast Asia by Alkem Company (S) Pte. Ltd., Singapore; in the English language in India, Bangladesh, Sri Lanka, Nepal, and Mauritius by Prism Books Pvt., Ltd., Bangalore, India; and in the English language in Japan by Toppan Co., Ltd., Tokyo, Japan.

Printed in the United States of America

Library of Congress Catalog Number: 00-060997

ISBN: 0-932633-41-2 12 11 10 9 8 7 6 5 4 3 2 1

DEDICATION

A wonderful friend and fine colleague of ours started this book: He laid out the basic ideas and the structure, and developed some of the materials. Then, suddenly and unexpectedly, and in the prime of his life, he passed away. We miss him sorely.

We are grateful to his brother Shams for providing us with the original materials, and honored at being given the opportunity to complete the project. It is not the same book it would have been had he completed it, but we hope it comes somewhere close to his expectations.

It is unusual to dedicate a book to one of its authors, but these are unusual and tragic circumstances. So, we humbly dedicate Process for System Architecture and Requirements Engineering *to the memory of a true visionary in the field of system development:*

Imtiaz Pirbhai.

Acknowledgments

We are deeply indebted to Kamal Hammoutene, Alan Hecht, Mark Maier, Vince Peterson, and Tanehiro Tatsuta for their painstaking reviews of the draft of this book and for their excellent comments and suggestions. The book is far better for their efforts.

We thank the editorial staff at Dorset House Publishing—Nuno Andrade, Debbie Carter, David Crohn, Wendy Eakin, Bob Hay, Mike Lumelsky, David McClintock, Matt McDonald, and Mike Richter—for the excruciating detail with which they examined, tore apart, and reconstructed our English. This, too, greatly improved the final product.

Finally, we thank the many participants in our seminars and workshops for their enthusiasm and for teaching us at least as much as we taught them.

Contents

Figures

Chapter 3

A Framework for Modeling Systems

3.1 A Model Framework

In the preceding chapter, we discussed numerous facets of systems in the categories of general characteristics, views, and requirements.

We now focus on the development of such systems. In this chapter, we explore the role of models in system development and introduce a framework to organize the many different models created during the development process. This framework captures what we know about systems, including the properties discussed in Chapter 2.

The framework serves as a road map for the development process, providing a cue for *what* models to build and *how* to build them. It assists us in keeping all the models related and linked to each other. In the first sections that follow, we start with a discussion of models and their usefulness.

3.2 Models in General

Most industries use models for purposes such as studying requirements for systems, examining feasibility and manufacturability, and determining how to build an actual system. In the computer hardware and software industry, models are used for some parts of the development process (usually for software requirements or design, or for hardware layout), but no techniques are widely used for modeling the *entire* system. Before discussing such modeling techniques, we discuss why models are useful.

3.2.1 Models Are Useful Abstractions

As Figure 3.1 shows, a model is an abstraction highlighting some aspects of real-world systems in order to depict those aspects more clearly. A model has an objective (the question we want it to answer) and a viewpoint (the point of view of one or more stakeholders: users, developers, and so on). Abstract models reduce the complexity of the real world to digestible chunks that are simpler to understand.

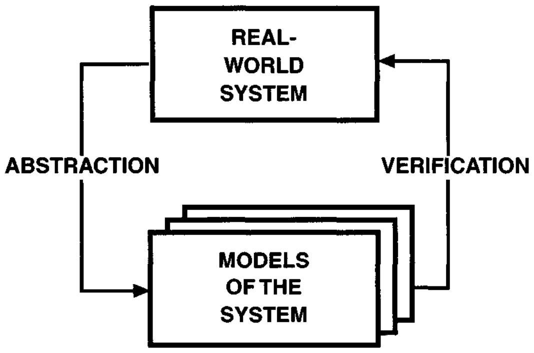


Figure 3.1: The Role of a Model.

On the other hand, abstract models are just representations, omitting some aspects of real-world systems, at least temporarily, but mapping what we hope to understand into a form that we can understand. Different types of models answer different types of questions about the system they represent, but even if we build a hundred different models, they could not answer every possible question about the system. That can only be done by the final system itself.

If we decide to build more than one model of a given system to investigate different aspects, then we should somehow organize these models according to their relationships to each other and to the system. This is why we need a framework.

3.2.2 Model Representations and Reuse

Before we discuss the framework for modeling systems, we expand on the idea of using models in two ways: First, models can be expressed using different notations; second, good models can be reused in different applications.

Though they appear very different from each other, Figures 3.2 through 3.5 can all represent the same scenario. Consider this description of a junior high school:

> Students enter school in seventh grade. Most of the students proceed to eighth grade, but some skip directly to ninth. Nobody graduates directly from eighth grade, but some leave school before graduating. The rest go on to ninth grade and then graduate.

In Figure 3.2, "The Bathtub Model"—adapted by permission from *General Principles of Systems Design* by Gerald M. Weinberg and Daniela Weinberg [Weinberg 88]—flows into and out of the various tubs can represent the flows of students into and out of grades. Tubs 1, 2, and 3 represent 7th, 8th, and 9th grades, respectively. S indicates the set of students entering school. P1 represents the students progressing to 8th grade. Q1 depicts the small number of students skipping 8th grade and going directly to 9th grade. P2 shows the normal progress from 8th to 9th grade. Q2 and Q3 show students leaving school without graduating. P3 represents the students graduating from 9th grade.

The Weinbergs use the bathtub model to explain the set of differential equations given here as Figure 3.3: Those equations can abstract the same junior high school situation in a different manner. N1, N2, and N3 either can represent the quantities of water in the three tubs or can indicate the number of students in three grades. N1' represents the rate of change of N1 over time, and so on.

Figure 3.4—a Structured Analysis data flow diagram—shows yet another representation of the bathtub model, and of the same real-world system.

Finally, Figure 3.5 gives the context diagram for Figure 3.4, once again representing the same real-world system, but in a more abstract form.

The models in Figures 3.2 through 3.5 also can show how models can be reused. The four models can be used to illustrate different applications, fitting the following description of a company's training program just as well as they fit the junior high school scenario, and just as well as they could fit many other similar scenarios.

Everyone joining Company X starts as an unskilled worker. The company's policy is to provide training and education for its employees. No one is allowed to work without some minimal vocational training that gets him or her into the semi-skilled labor pool. Those who have college degrees move to the skilled category, bypassing the semi-skilled pool. After five years in the semi-skilled category, workers automatically progress to the skilled pool. Eventually, employees either leave for better opportunities or retire.

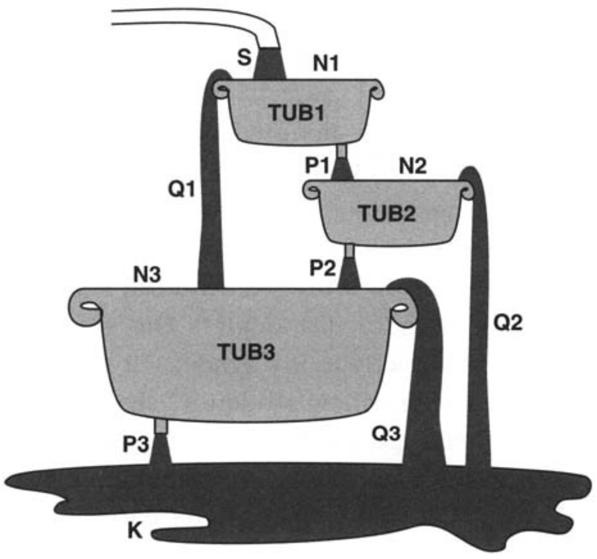


Figure 3.2: The Bathtub Model.

$$N1' = S - (P1 + Q1)$$
$$N2' = P1 - (P2 + Q2)$$
$$N3' = (Q1 + P2) - (P3 + Q3)$$

Figure 3.3: Equations Representing the Bathtub Model.

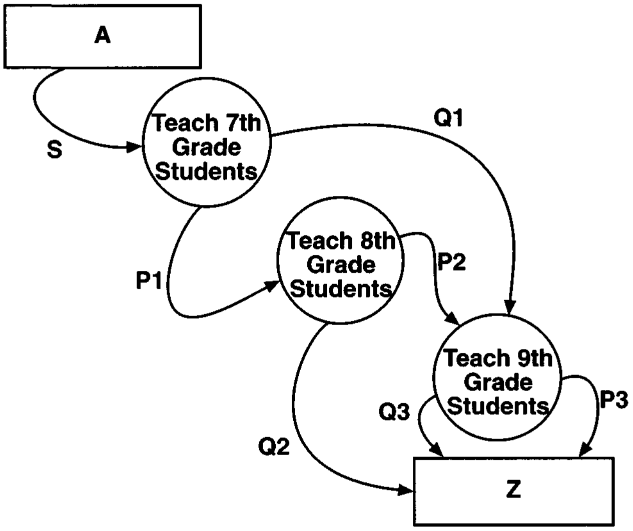


Figure 3.4: A Data Flow Diagram Representing the Bathtub Model.

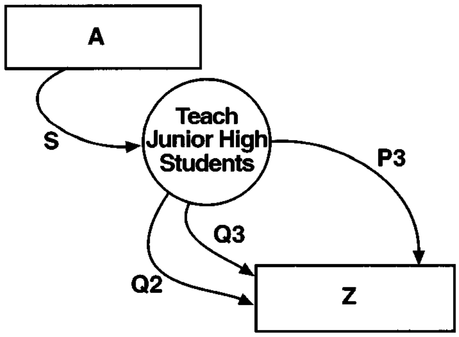


Figure 3.5: A Context Diagram of the Model in Figure 3.4.

3.3 Exploiting System Hierarchies

In Sections 2.1.2 and 2.1.3, we explained that all real-world systems consist of subsystems, or—looking in the other direction—that every system is part of a larger system. In other words, systems come in hierarchies. Using these hierarchies is the first step in constructing our modeling framework.

3.3.1 Why Exploit Hierarchies?

Why do we want to exploit the idea of system hierarchies? Because we want to reduce complexity by not thinking about everything at once.

- At the highest level of a model, we establish the place of the system in its environment and define the broad objectives of the system and its relationships with that environment (for example, communication and physical linkage).
- Using these broad objectives, we proceed into the requirements and architecture of our system, remembering that it can be manual, automated (by various technologies), or both. Creating an architecture for the system partitions it into subsystems that can themselves be considered self-contained systems—similar to the top-level system. By iterating this partitioning procedure, as illustrated in Figure 3.6, we simplify the problem by treating each subsystem, sub-subsystem, and so on, as a system in its own right, with its external interconnections and interactions represented in the level above.

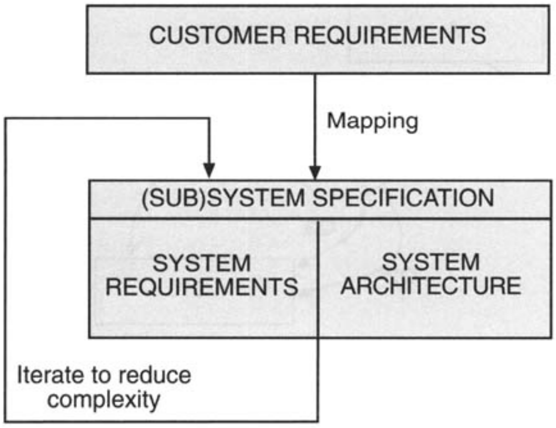


Figure 3.6: Hierarchies to Reduce Complexity.

3.3.2 What Are the Benefits and Pitfalls of Layered Systems?

There are many benefits in hierarchically organized systems and subsystems. Every layer of system definition supplies some of the requirements for the layer below. At the top, a firm link is established between the system and its environment. If we can stabilize the upper-level requirements and architecture early, the lower-level design can proceed much more effectively. We can anticipate high-risk subsystems and use prototyping to resolve those risks. Working on a certain level of abstraction helps us concentrate on that level and not get too detailed too fast.

One point needs to be strongly emphasized:

> System specification and development are not necessarily top-down processes.

Overlooking this heuristic can be a major pitfall. The fact that, for convenience, many of our descriptions of the process are presented top-down does not detract from this statement; neither does the top-down appearance of the figures. The top layers do not have to be complete before we can work on the lower layers; in some cases, it is appropriate to work upward from the lower layers. Think of development as a concurrent or iterative process—there is always some work going on in every layer.

The layered model results in a specification hierarchy and a representation of the requirements flows between layers. The process of filling in the framework and developing these models is discussed in Chapter 5, and illustrated in Part II.

Integration is key to developing the system: "The whole is greater than the sum of its parts." The systems we develop require that all of their components are integrated: software with software; hardware with hardware; software with hardware; automated with manual; and especially, system with environment. So, despite developing many separate models, we need subsystems linked to other subsystems, and layers linked to higher and lower layers. This is the purpose and benefit of the modeling framework.

3.3.3 How Many Models?

Many methods insist on building one large analysis model (as in Structured Analysis or entity-relationship modeling) and, separately, one large design model (as in Structured Design or in many object-oriented methods). We, too, use the "divide and conquer" approach in our framework, but we ensure that the submodels are integrated.

How many models do we build, then? If we consider an integrated set of submodels as a single model, then we build a model for the overall system and a model for each of its subsystems, sub-subsystems, and so on. Each of these models itself can contain numerous models of its own: models for requirements, design, architecture, information structures, interconnects, and many more. Good methods, and tools that automate them, will support all of these multifarious models and the links between them.

3.3.4 Where Do We Stop?

We have established that every system is part of a larger system. Looking in the other direction, How far down should we decompose a subsystem into further subsystems?

In larger systems, system developers will stop at the level where direct system responsibility ends, or where they have no constraints to impose internally to a subsystem. Then, specialists in those subsystems can decide whether to continue with the same process or to switch to some other approach that is specific to their discipline. For example,

- we decompose a multidisciplinary system into parts that are, say, mechanical or hydraulic, and pass those subsystems to the corresponding specialists
- in MIS, we often decompose until we can clearly differentiate between human activities (for example, clerks doing part of the work) and software activities (computer programs doing the remaining work)
- in embedded, real-time systems, we might decompose until we have a better understanding of the hardware/software split

A software subsystem can usually be decomposed into further subsystems, and an organizational subsystem can be organized as cooperating groups of further organizational subsystems. The techniques, methods, and tools for specialized subsystem development are often more mature and better automated than those used for overall system development. In this book, we do not discuss specialized hardware, software, or organizational methods, but instead refer to other publications on these topics.

On the other hand, we do not have to decompose every system into subsystems that comprise only one technology. Sometimes, the system levels stop where several related technologies are used in a single subsystem. For example, in a hydraulic subsystem with electromechanical valves, it would not make much sense to separate these two technologies, because they exist to support each other.

Figure 3.7 shows various alternatives for decomposing subsystems: The top-level decomposition separates a human subsystem from a purely mechanical subsystem and leaves a multi-technology subsystem to be further decomposed. On the next level, a software subsystem is further decomposed into two software subsystems. At the lowest level, we find a software subsystem, a human subsystem, and a subsystem that uses mixed technology but is treated as one unit.

In later chapters, we discuss more criteria to determine where to stop decomposing.

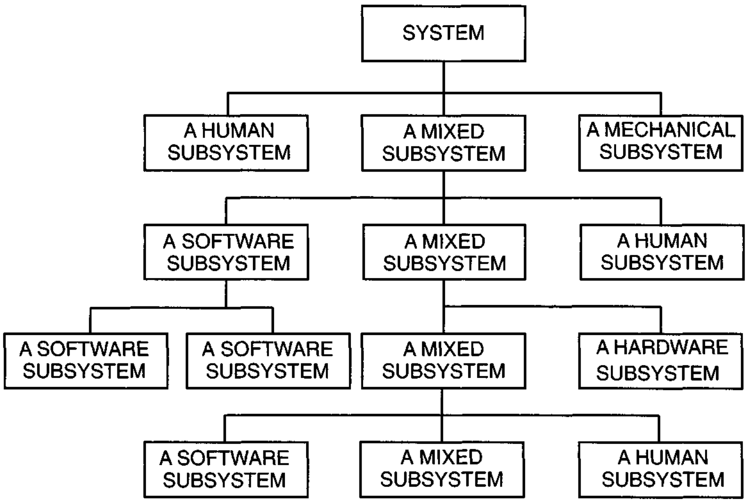


Figure 3.7: Partitioning a System.

3.4 Exploiting the *What/How* Classification

The next step in constructing our modeling framework is to use the *what/how* classification of systems that we discussed in Section 2.2.3. As you may have noticed in Figure 3.6, every specification consists of two parts: system requirements and system architecture. Both of these parts contain models. The system requirements model is a technology-independent model of the problem the system is to solve: It represents the *what*. The system architecture model is a technology-dependent model of the solution to the problem: It represents the *how*. These two models are created for the entire system and for every subsystem—hardware, software, human, or mixed technology—down to the lowest level.

3.4.1 Separation of *What* and *How*

The separation of the *what* and the *how* is extremely important for the following (and possibly other) reasons:

- It is often very useful to understand a problem (the *what)* independently of any particular solution (the *how).* (Conversely, there are situations where it is useful to develop a single architecture that will satisfy a whole class of problems.)
- Any given problem has many possible solutions. Selection of a particular solution (the *how)* is a trade-off process; we often need to make numerous different trade-offs while keeping the problem statement (the *what)* unchanged.
- The separation supports the generally recognized principle of separation of concerns, which means dealing with only one part of the system's complexity at a time. The requirements model (the *what*) only has to cope with essential problems; the architecture model (the *how*) has to cope with many constraints imposed by technology, organization, and so forth.
- Finally, seldom do we build systems totally from scratch. Most systems we build are either implementations using new technology (only changing the *how)* or the integration of several previous systems into a new system.

The separation of the *what* and the *how* gives us the power to reimplement the *what* using new technology, but it also gives us the power of reusability—not just for software or hardware, but for requirements as well. This is particularly important, because requirements are much more stable over much longer periods of time than technology [McMenamin 84].

In this book, we use the *what/how* classification for yet another important purpose. As we construct several different models later in the book, we would have to handle a lot of complexity at once if we addressed how to construct them at the same time as we addressed what to construct. So, we have split the following chapters using the *what/how* separation: Chapter 4 describes what models belong in our framework; Chapters 5 through 7, and all of Part II, describe how to develop these models.

As mentioned, Chapters 5 through 7 describe the *how* of constructing the models, but in Part II, we describe the *how* of applying these models to some real systems—from that perspective, Chapters 5 through 7 represent the *what!* This is analogous to the layered structure of systems, in which architectural or design decisions in one layer result in requirements in the layer below. This same principle applies to this book, which itself is a kind of system.

The purpose of Part II and of the on-line model is to exemplify what real projects must do—the *what*, the *how*, and also the *when*. The *when* refers to project planning and scheduling, including such issues as which tasks are conducted concurrently, and which sequentially. Throughout the 1970's and 1980's, simplistic process models like the waterfall model predominated. We know now that there are no simple solutions to project planning and scheduling. Rather, these are decisions that must be made for each project, by project management. It is the manager's job to observe the process, to watch and interpret the results of individual steps, to take into account many constraints, and based on all of that, to reconfigure dynamically the *what*, the *how*, and the *when* of the project.

3.4.2 The Architecture Template

Our modeling framework employs an extension of the *what/how* split to classify systems, subsystems, their components, and their activities according to a generic architecture template. Figure 3.8 shows that this architecture template classifies a system or subsystem into five categories:

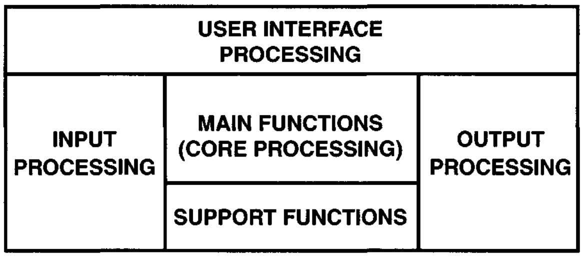


Figure 3.8: The Architecture Template.

1. The center region contains the main functions of the system: the core functional processing. Here, we model things that the system absolutely has to do, things that belong to the essence of the system, independent of any technology.

2. The top region hosts those parts of a system that interact with the users. It contains all subsystems, functions, and activities that make up the human-machine interface. It controls access to the system, and it accepts input from and prepares output for the human user, all in whatever forms are established with the user.
3. The left region contains the functions and subsystems that interface with other systems and subsystems to provide input for our system. It has to establish interconnections, request input, check it for acceptability, preprocess it, and perform many other input-related activities.
4. The right region provides similar resources for the output of our system to other systems. This includes establishing interconnection, converting output to the form needed for transfer, sending it, and so on.
5. The bottom region houses any functions or subsystems that provide support to the rest of the system to keep it running. These include self-test procedures, error logging, fault detection, and also maintenance functionality. This region is fundamentally different from the other outer regions: those regions deal with various kinds of interfaces between the system and its environment, whereas the bottom region deals with functions that support the system internally. The support functions might require additional inputs and outputs, but still, they are internal functions.

How do we use the architecture template in our modeling framework? As shown in Figure 3.9, the template mainly helps us with mapping between requirements (the *what)* and architecture (the *how).* This mapping can be applied on any level: for the overall system and for every subsystem on any layer.

From the requirements viewpoint, we augment or enhance the required functionality of the system (which is modeled in the core part of the template) with a ring of functionality supporting the core processing of the system. These augmented, or enhanced, requirements are packaged into architectural subsystems.

From the architecture viewpoint, the template provides an excellent starting point for building information-hiding subsystems [Parnas 71]. The center hides the essential functions, the top region hides the user-interface technology and behavior, the left and right regions hide input/output specifics, such as device characteristics and protocols, and the bottom region hides support functionality, such as service and maintenance modules, and many more. An alternative name for the template might be the information-hiding template.

Once we establish the functionality of the system and subsystems, we can easily categorize and extract the core requirements for future reuse.

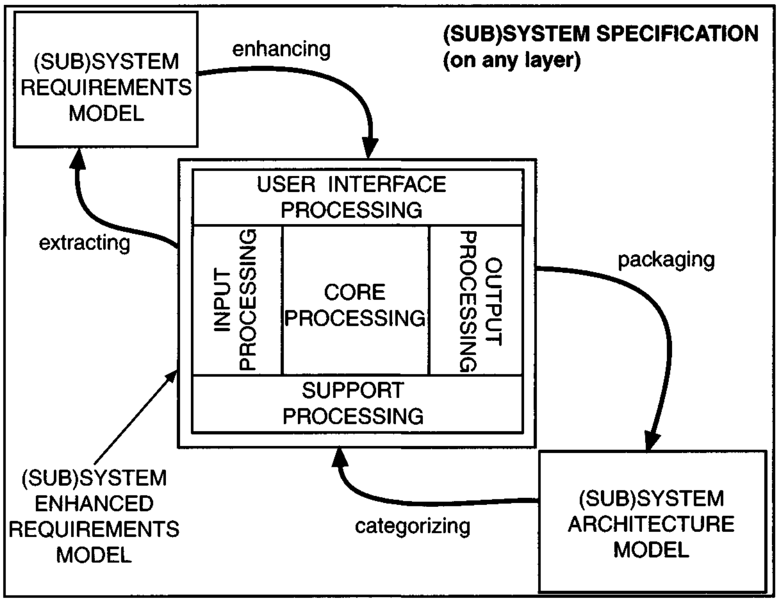

(SUB)SYSTEM REQUIREMENTS MODEL
enhancing
(SUB)SYSTEM SPECIFICATION (on any layer)
extracting
USER INTERFACE PROCESSING
INPUT PROCESSING
CORE PROCESSING
OUTPUT PROCESSING
SUPPORT PROCESSING
packaging
(SUB)SYSTEM ENHANCED REQUIREMENTS MODEL
categorizing
(SUB)SYSTEM ARCHITECTURE MODEL

Figure 3.9: The Template Bridging Requirements and Architecture Models.

3.4.3 Using the Architecture Template

In this section, we present two examples of using the architecture template to classify the functionality of systems. To demonstrate that it can be used on any layer of a system hierarchy and for any kind of application (automated or not), we start with an organizational system composed of automated and manual parts.

The template in Figure 3.10 illustrates the activities that are performed by nurses at their station in a hospital. We can divide the activities into the five categories. By doing so, if the procedures for helping visitors (in the user-interface part of the template) are changed, the rest of the system can remain unchanged. The same is true for changing the policy for checking stock supplies in the maintenance region, and for any changes involving just a single region of the template.

USER INTERFACE	• HELP VISITORS • GUIDE PATIENTS TO DOCTORS' OFFICES • ANSWER PHONE CALLS • RESPOND TO PATIENT EMERGENCY CALLS	
INPUT PROCESSING	***MAIN FUNCTION***	***OUTPUT PROCESSING***
• CHECK IN PATIENTS • SCHEDULE PATIENT MONITORING • RECEIVE AND LOG INCOMING MEDICATION • RECEIVE SUPPLIES	• MONITOR PATIENT HEALTH • DISPENSE PRESCRIBED MEDICATIONS • ADMINISTER ROUTINE TESTS • KEEP PATIENT HEALTH RECORDS • KEEP PATIENT BILLING RECORDS • ASSIST DOCTOR ON ROUNDS	• CHECK OUT PATIENTS • ISSUE ALERTS TO DOCTORS AND STAFF • FILE REPORTS • ORDER SUPPLIES • SEND PATIENT BILLING TO ACCOUNTING DEPARTMENT
	SUPPORT	
	• MONITOR DESK OPERATIONS • CHECK NURSES' SCHEDULES • CHECK STOCK SUPPLIES	

Figure 3.10: Classification of Activities in a Manual System.

This nurses' station is a subsystem of an overall hospital system, for which we could also classify the activities. Depending on the purpose of the whole hospital model, the nurses' station would be part of the system core or may be considered part of the maintenance subsystem (to support the doctors).

The second example is of an automated system. Using the hospital application, we show how to classify the automated activities of a patient-monitoring system in Figure 3.11.

The architecture template is a very powerful modeling tool that we use repeatedly to classify system or subsystem activities. Not only can it be used to bridge the requirements/architecture models, it can be used very early in a project, before we even know the requirements or make decisions on the architecture, to discover topics to be treated in more detail later. It can also be used in distributing work among project members, allowing them to work concurrently.

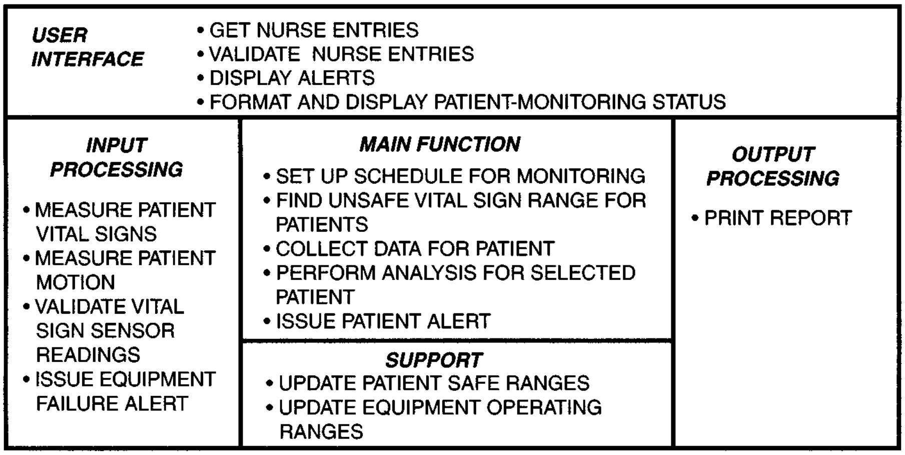

USER INTERFACE
• GET NURSE ENTRIES
• VALIDATE NURSE ENTRIES
• DISPLAY ALERTS
• FORMAT AND DISPLAY PATIENT-MONITORING STATUS
INPUT PROCESSING
• MEASURE PATIENT VITAL SIGNS
• MEASURE PATIENT MOTION
• VALIDATE VITAL SIGN SENSOR READINGS
• ISSUE EQUIPMENT FAILURE ALERT
MAIN FUNCTION
• SET UP SCHEDULE FOR MONITORING
• FIND UNSAFE VITAL SIGN RANGE FOR PATIENTS
• COLLECT DATA FOR PATIENT
• PERFORM ANALYSIS FOR SELECTED PATIENT
• ISSUE PATIENT ALERT
SUPPORT
• UPDATE PATIENT SAFE RANGES
• UPDATE EQUIPMENT OPERATING RANGES
OUTPUT PROCESSING
• PRINT REPORT

Figure 3.11: Classification of Activities in an Automated System.

3.5 Exploiting the Information/Material/Energy Classification

The final step in constructing our modeling framework is to classify systems according to their information, material, and energy processing characteristics that we discussed in Section 2.2.1.

3.5.1 A Generic Subsystem Structure

In Chapter 2, we showed that everything a system does can be classified into material processing, energy processing, and information processing. Since material and energy processing are quite different from information processing, we can treat these two areas separately. If we combine this decision with the idea of categorization provided by the architecture template, we end up with the system partitioning shown in Figure 3.12, which divides the processing into finer classifications. These finer classifications become subsystems of the overall system. As Figure 3.12 shows, there are subsystems that do the two different types of processing, but the overall, or boundary, subsystem does both. Figure 3.13 explains more about the functions of the various subsystems.

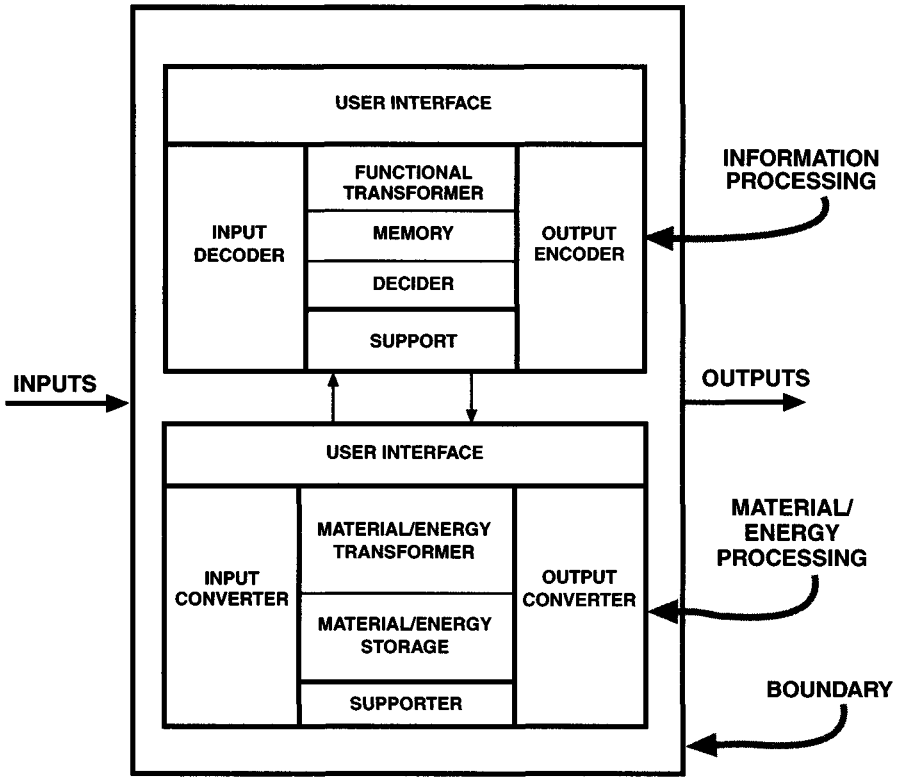


Figure 3.12: Separating Material/Energy Processing from Information Processing.

Note that there is no box in material/energy processing equivalent to the decider in information processing. With today's systems, the decision-making function is almost always an information processing function. Consequently, Figures 3.13 through 3.16 all have a blank entry in the material/energy processing side.

Boundary: Subsystem(s) that form a barrier around a system, shielding it from its environment.	
Information Processing Subsystems	**Material/Energy Processing Subsystems**
User Interface: Subsystem(s) to allow information exchange with external human users.	**User Interface:** Subsystem(s) to allow material/energy exchange with external human users.
Input Decoder: Subsystem(s) to convert the coding of external information for internal use.	**Input Converter:** Subsystem(s) to transform material/energy from external to internal forms.
Functional Transformer: Subsystem(s) to transform input information into output information.	**Material/Energy Transformer:** Subsystem(s) to transform and associate material/energy inputs to outputs.
Memory: Subsystem(s) to retain for later use information, its relationships, and its organization.	**Material/Energy Storage:** Subsystem(s) to store material/energy for later use.
Decider: Subsystem(s) to control (for example, enable, inhibit, or trigger) functional transformers.	
Output Encoder: Subsystem(s) to convert the coding of internal information for external use.	**Output Converter:** Subsystem(s) to transform material/energy from internal to external forms.
Support: Subsystem(s) to support system monitoring, servicing, and reconfiguration.	**Supporter:** Subsystem(s) to enable maintenance, growth, and reconfiguration.

Figure 3.13: Generic Description of Subsystem Responsibilities.

3.5.2 Categories of a Deliverable System

For a deliverable system, product, or component, the categorization scheme introduced above can be a useful starting point for brainstorming the subsystems. Figure 3.14 makes the generic categories of Figure 3.13 specific to all deliverable systems, and Figure 3.15 makes them specific to a cruise control system.

Note that the categories in the generic template are applicable to many systems, although there are usually some that are not applicable to a specific system. Consider the generic categories a pattern for thinking about a system and its subsystems.

Boundary: The external housing, casing, or such other exterior that shelters the system from its environment.	
Information Processing Subsystems	**Material/Energy Processing Subsystems**
User Interface: Data entry and display devices.	**User Interface:** Access mechanisms allowing operator insertion and extraction of physical items and electrical or mechanical energy.
Input Decoder: Processor(s) of information inputs from other systems, converting them, as needed, from their received formats to internal system formats.	**Input Converter:** Mechanism(s) for the physical manipulation of received physical items or energy into the orientation or form needed internally.
Functional Transformer: Input-to-output information conversion devices performing processes such as algorithms, functions, math equations, or string manipulations.	**Producer:** Electrical or mechanical devices that process received physical items or energy, and convert them into the desired product. For example, an automatic mechanism that receives component parts and assembles them into a finished product.
Memory: Device(s) that store, for later use, information from the operator, from other systems, or from the processes of this system retained, possibly with its relationships and organization.	**Storage:** Any part(s) of the system that store material or energy for later use, such as a storage room, a shelf, a battery, or a water reservoir.
Decider: Control processors by which information processing and resources are scheduled, and which establish the different states or modes of behavior of the system.	
Output Encoder: Processor(s) of information outputs to other systems, converting them, as needed, from their internal system formats to external formats.	**Output Converter:** Mechanisms for the physical manipulation of produced physical items or energy into the orientation or form needed externally. For example, the automatic packaging of manufactured products for shipment.
Support: Processors that perform tasks such as fault isolation, error handling, service monitoring, system reconfiguration, and graceful degradation.	**Supporter:** Access mechanisms for physical maintenance, growth, and reconfiguration.

Figure 3.14: A Template for Subsystems of a Deliverable, Human-Operated System.

Boundary: The external housing, casing, or such other exterior that shelters the system from its environment.	
Information Processing Subsystems	**Material/Energy Processing Subsystems**
User Interface: Data entry and display devices.	**User Interface:** Access mechanisms allowing operator insertion and extraction of physical items and electrical or mechanical energy.
Input Decoder: Processor(s) of information inputs from other systems, converting them, as needed, from their received formats to internal system formats.	**Input Converter:** Mechanism(s) for the physical manipulation of received physical items or energy into the orientation or form needed internally.
Functional Transformer: Input-to-output information conversion devices performing processes such as algorithms, functions, math equations, or string manipulations.	**Producer:** Electrical or mechanical devices that process received physical items or energy, and convert them into the desired product. For example, an automatic mechanism that receives component parts and assembles them into a finished product.
Memory: Device(s) that store, for later use, information from the operator, from other systems, or from the processes of this system retained, possibly with its relationships and organization.	**Storage:** Any part(s) of the system that store material or energy for later use, such as a storage room, a shelf, a battery, or a water reservoir.
Decider: Control processors by which information processing and resources are scheduled, and which establish the different states or modes of behavior of the system.	
Output Encoder: Processor(s) of information outputs to other systems, converting them, as needed, from their internal system formats to external formats.	**Output Converter:** Mechanisms for the physical manipulation of produced physical items or energy into the orientation or form needed externally. For example, the automatic packaging of manufactured products for shipment.
Support: Processors that perform tasks such as fault isolation, error handling, service monitoring, system reconfiguration, and graceful degradation.	**Supporter:** Access mechanisms for physical maintenance, growth, and reconfiguration.

Figure 3.15: Subsystems of a Cruise Control System.

3.5.3 Categories of a People System

The systems we build reflect the organizations that build them. With this in mind, we can devise a categorization of organizational systems, such as of whole companies, departments, or groups of people cooperating to achieve a certain goal. Figure 3.16 maps the generic categories onto an organizational structure describing the various subsystems in such a context. Figure 3.17 shows categories specific to a garment factory. This kind of categorization helps to distinguish between value-adding functions and overhead functions in an organization: It can be used as a starting point for modeling business processes and identifying essential parts of them.

3.6 Layered Models: The Truth at Last!

We discussed numerous characteristics and features of models in the previous sections, but we kept the important issue of layered models for the end of this chapter. In any discussion of systems, models of systems, or the process of building systems, the term "layer" plays an important role. Here, we explore several unique aspects of layers, and of the different relationships between layers, between elements in one layer, and between elements in different layers.

One purpose of this section is to dispel a couple of myths. First, there is the myth that all layered models fall into the category of functional decomposition or, worse yet, top-down functional decomposition. And second, that layered models are fundamentally incompatible with object orientation.

In the first pages of *Strategies for Real-Time System Specification,* we introduced a diagram titled, "The Total System Life Cycle," little realizing at the time just how significant it was. It showed various layers of the system modeling process and layers of specifications resulting from that process. We elaborate on that diagram in Figure 5.1 of this book, but for now, we discuss some of its implications. What we have realized since creating that diagram is that there is tremendous similarity between systems, system models, and the system development process. Layers are an important part of these similarities, but they are also the source of some confusion. There is not just one kind of relationship between layers or elements of layers: We can identify several basic relationships that keep recurring in different systems, system models, and in the development process.

Boundary: The external housing, casing, or such other exterior that shelters the system from its environment.	
Information Processing Subsystems	**Material/Energy Processing Subsystems**
User Interface: Data entry and display devices.	**User Interface:** Access mechanisms allowing operator insertion and extraction of physical items and electrical or mechanical energy.
Input Decoder: Processor(s) of information inputs from other systems, converting them, as needed, from their received formats to internal system formats.	**Input Converter:** Mechanism(s) for the physical manipulation of received physical items or energy into the orientation or form needed internally.
Functional Transformer: Input-to-output information conversion devices performing processes such as algorithms, functions, math equations, or string manipulations.	**Producer:** Electrical or mechanical devices that process received physical items or energy, and convert them into the desired product. For example, an automatic mechanism that receives component parts and assembles them into a finished product.
Memory: Device(s) that store, for later use, information from the operator, from other systems, or from the processes of this system retained, possibly with its relationships and organization.	**Storage:** Any part(s) of the system that store material or energy for later use, such as a storage room, a shelf, a battery, or a water reservoir.
Decider: Control processors by which information processing and resources are scheduled, and which establish the different states or modes of behavior of the system.	
Output Encoder: Processor(s) of information outputs to other systems, converting them, as needed, from their internal system formats to external formats.	**Output Converter:** Mechanisms for the physical manipulation of produced physical items or energy into the orientation or form needed externally. For example, the automatic packaging of manufactured products for shipment.
Support: Processors that perform tasks such as fault isolation, error handling, service monitoring, system reconfiguration, and graceful degradation.	**Supporter:** Access mechanisms for physical maintenance, growth, and reconfiguration.

Figure 3.16: A Template for an Organization That Builds Deliverable Systems.

Boundary:
The building(s) that house the garment factory and its offices.

Information Processing Subsystems	Material/Energy Processing Subsystems
User Interface: People who negotiate the designs and contracts with customers.	**User Interface:** Direct factory sales outlet, returns department.
Input Decoder: Purchase order processing, inventorying of received materials.	**Input Converter:** People unpacking received materials, repackaging them in a form that supports the production process; people from personnel department screening new hires.
Functional Transformer: Design department and equipment that transforms customer requests into actual designs to be manufactured.	**Producer:** Production line personnel and equipment that convert the received materials into finished garments according to the selected designs.
Memory: Storage of the designs, accounting records, shipping records, the employee records, and so on.	**Storage:** Supply cabinets, storage lockers, stock room, and so on.
Decider: Management that determines the production schedules, the factory plans, and the coordination of the factory floor.	
Output Encoder: People responsible for invoicing the customers, for waste disposal coordination, and so on.	**Output Converter:** The shipping department, delivery truck drivers, and so on.
Support: Customer accounting, payroll department, and so on.	**Supporter:** The facilities and maintenance crews who remove trash and keep the factory in operating condition.

Figure 3.17: Subsystems of a Garment Factory.

Systems, models of systems, and the system development process share the following attributes:

- They are layered.
- The layers—once they are identified—form a structure that can be read and interpreted in any sequence: from the top layers to the lower layers, from right to left, from bottom to top, and so on. Moreover, the layers can be *developed* in any sequence: top to bottom, right to left, bottom to top, and so on, and the interpretation and development sequences are quite independent of each other.
- The number of elements per layer typically increases downward, giving the whole structure a pyramidal shape; but note that we sometimes have an independent structure of elements within one of the main layers. For such a structure, the basic statement of this paragraph is still true: The number of elements tends to increase downward.
- The elements forming the layered structure can be considered a set, either of activities or of entities. In any particular system, system model, or development project, these elements may be carried out or used in some prescribed sequence, concurrently, or in any combination of sequence and concurrency.
- Elements in the layers usually communicate and cooperate up, down, and sideways within and between layers. Communication and cooperation can be in the form of information, material, or energy, depending on the kind of system, model, or process in question. Some earlier models restricted the development process by asserting, for example, that all information flows vertically through the top layer, and that only information (not material or energy) can be communicated with the outside world. However, our model recognizes that information, material, and energy can all flow sideways to and from individual layers, and can all interact with the outside world.
- Every layer includes, deals with, or is associated with, some requirements, some architecture or design, some construction or implementation, and some integration and testing. Also, each layer usually requires planning, quality assurance, management, and other items, but we are not addressing these in this book.

An important conclusion for us is that layered models—for systems or as meta-models for system development processes—do not inherently imply any particular sequence. They represent a static structure (of a system, its development, the development process, or models of any and all of these) that can be populated in any convenient sequence that makes sense for the problem at hand. This point was beautifully made by Parnas—arguably the father of information-hiding structures—in [Parnas 86].

So, the layers and the elements in layers are nondirectional, but we are interested in their relationships. There are probably many types of relationships in layered models, but four of them are of special interest in system development: aggregation/decomposition, abstraction/detailing, supertype/subtype, and controlling/controlled. Let us look at each of these in detail, discussing properties of their relationships and examples from our methods and other well-known approaches.

3.6.1 Aggregation/Decomposition Relationship in Models

The architecture model, resulting from the architecture method, is an example of an aggregation/decomposition model. Such models characterize real physical elements, their sub-elements or parts, and their super-elements or assemblies. Elements in the higher layers actually consist of the elements in the lower layers, or conversely, elements in the lower layers are decompositions of those in the higher layers. The structure is also known as a whole/part structure [Coad 91] or a container/content structure: A given layer provides the container for the layer below, which is the content of the layer above. In entity-relationship modeling, entities can be linked by composed-of or consists-of relationships. In manufacturing terms, it is an assembly/subassembly/component structure. This type of structure is pervasive in engineering and in everyday life.

An aggregate actually involves more than just collecting sub-elements into a set. The sub-elements must also interface with each other, requiring linkages between them that may not be evident when they are considered separately. This is why we discuss enhancement of abstract requirements, using the architecture template, in our methods when the requirements are mapped into real physical modules.

We can better imagine aggregation/decomposition structures applied at the system levels, where physical hardware of various kinds is involved. For software, which does not have a physical form, it is not so clear. The trick is to imagine that software does have a physical form. A complete software program or assembly can be considered as an architecture module at the highest software layer; major subprograms it contains are modules in the next layer down; sub-subprograms or subroutines (if any) form a further layer, and so on. As we discuss elsewhere in this book, a transition can be made from an aggregation/decomposition model to an object-oriented representation by defining modules to be aggregate objects, as described, for example, in [Page-Jones 95, Section 4.2]. Once in the object-oriented domain, other structures may apply, depending on the particular object-oriented approach used.

To summarize the usage of this relationship: We build layers to show physical packaging of elements into larger groups or assemblies. In each layer, we can define physical interfaces between elements or between groups. The grouping forms a sort of fence around its elements, potentially protecting the visibility of the interior elements or regulating the access to them. In software development, we use terms like information hiding, scope, and visibility control to describe the nature of the aggregation/decomposition relationship.

3.6.2 Abstraction/Detailing Relationship in Models

When we use an abstraction/detailing relationship in models, the higher layers are simply more abstract expressions of the lower layers, or conversely, the lower layers are more detailed expressions of the higher layers. The most familiar example of this relationship occurs in Structured Analysis (SA), usually represented by data flow diagrams. (Note that the control model of the real-time extensions of SA does not use this relationship—see Section 3.6.4.) The process model part of the requirements model, being founded on SA, uses the abstraction/detailing relationship for processes and their child diagrams. In a sense, the whole requirements model—if applied correctly—is abstract throughout because it consists only of narrative statements (albeit in structured form) that do not necessarily correspond to real physical groupings of processes, entities, or control structures.

The abstraction/detailing relationship often has been erroneously named abstraction/decomposition. Although we, too, have been guilty of using this terminology, we now disagree with it. First, abstraction and decomposition are not opposites, and the essence of these relationship name pairs is that they should be opposites, reflecting the upward and downward viewpoints in a layered model. Second, decomposition *is* the opposite of aggregation, which is why we use it in the aggregation/decomposition relationship described above, where something is broken into the elements it contains. Consider this: An abstract requirement statement does not contain the more detailed requirements statements that describe it; however, a physical system element does contain the separable subelements of which it is an aggregate. If we take the elements of a physical system and assemble them, we get the physical system; if we assemble a set of detailed requirements, we merely have a collection of detailed requirements—the abstract and detailed requirements exist independently of each other, with an abstraction/detailing relationship between them.

Our categorizations of layered structures, then, have led us to an interesting paradox. The terms to which we objected earlier—functional decomposition and top-down functional decomposition—are frequently applied to Structured Analysis and its data flow diagrams, yet data flow diagrams, when used correctly to represent abstract requirements, do not involve decomposition at all: They involve detailing. When we use Structured Analysis to create essential models according to our own guidelines and those in [McMenamin 84] and [Robertson 98], we are not decomposing downward through the layers; we are adding detail. Going upward, we are not packaging or aggregating; we are abstracting.

Of course, if you are misusing SA to represent the aggregation and decomposition of physical structures, then anything goes, and we cannot take responsibility for the results (which are usually awful).

How do detailing of the required capabilities and decomposition of the physical structure relate to each other? As a system is developed, they proceed in parallel, with sufficient detail added to the required capabilities to satisfy the needs of a particular physical layer. This point is illustrated further in Part II.

3.6.3 Supertype/Subtype Relationship in Models

In the supertype/subtype relationship, an element in the higher layer—the supertype—includes all of the features that are common to its associated elements in the lower layer—its subtypes. These features—in the simplest case—are attributes (as they are called in information modeling) that are inherited by the elements on the lower layer. Starting from the lower level, supertypes are formed for sets of elements that share common attributes. Thus, we might have at the top level "vehicle," and at the level below "ship," "aircraft," and "land vehicle." Below "land vehicle," we might have "bicycle," "motorcycle," "ATV," and "automobile." This tells us, for example, that an automobile is a land vehicle and a land vehicle is a vehicle.

Supertype/subtype models are important in object orientation. This relationship is the foundation for inheritance—one of the essential and most powerful features of object orientation. Attributes of "vehicle," in the above example, are inherited by all the other elements, and attributes of "land vehicle" are inherited by all of the elements in its subtypes. Object orientation has taken this relationship and extended it to more complex forms of inheritance than just attribute inheritance: The lower layer may also inherit functions (or operations, or "methods," as they are sometimes called) and the behavior of the supertypes.

Supertype/subtype relationships are also referred to as generalization/specialization relationships, class hierarchies, inheritance structures, and "is-a" hierarchies. With the supertype/subtype relationship, it is important that the supertype contains all the commonalities of the subtypes. The main use of this relationship is to discover commonalities and to describe them only once, thus reducing redundancy. The structure then allows the lower layers to inherit whatever commonalities have been discovered.

Now that we have defined the supertype/subtype relationship, we can see that the relationship is, in fact, a subtype of the abstraction/detailing relationship. A supertype is an abstraction of its subtypes, and the subtypes are detailed instances of the supertype. So, all supertype/subtype relationships are also abstraction/detailing relationships, but the converse is not true: Not all abstraction/detailing relationships are supertype/subtype relationships, because not all abstraction/detailing relationships follow the "is-a" principle. For example, a process on a data flow diagram and its child diagram are an abstraction/detailing pair, but it is not true that a child diagram "is-a" parent process.

3.6.4 Controlling/Controlled Relationship in Models

This relationship distinguishes between up and down by having the upper layers control elements of the lower layers. Other terms used for this relationship are the control hierarchy, or the is-boss-of/is-supervised-by relationship. Sometimes, we simply say that the higher element uses the lower elements. The higher layer must have knowledge of the lower layer but the lower layer—that is, the one being used—does not necessarily have to know anything about the boss. In terms of client/server models, the client is the boss that delegates work to the server; the server provides certain services that are performed whenever a client asks for them.

In the requirements method, the control model controls processes in the process model, by activating or deactivating them. In the architecture method, we can model client/server behavior to avoid iterative cycles between architecture modules.

Structured Design, the software design method, provides another example of a controlling/controlled relationship. In its main graphical model, the structure chart, a given module invokes (that is, it uses, calls, or controls the execution of) modules in the layer below. Structured Design is one of several methods that can be used in conjunction with the requirements and architecture methods, as described in *Strategies for Real-Time System Specification*, Section 24.3, and in Chapter 4 of this book.

3.6.5 Layered Models Summary

We hope we have succeeded in dispelling the myth that all layered models are built top-down using decomposition, and have shown that this simplistic, one-size-fits-all view of these models is wrong. The four types of relationships in layered models, described above, are distinctly different from each other; they all serve distinct and important roles in system development; and they can be integrated smoothly, where appropriate, with other models, including object-oriented models. Figure 3.18 summarizes the key aspects of the four relationships in and between layers or their elements.

Even though the four relationships are different, it is convenient to have at least one terminology that can be used with all of them. For this purpose, the "family tree" relationship analogy—of parent/child, grandparent/grandchild, and ancestor/descendant—is commonly used. Although close inspection shows that the analogy does not really fit all four of the relationships (for example, children are not decompositions of their parents), these terms sufficiently describe above/below relationships.

We can now enlarge on the statements of Section 2.1.3 "Multiple Hierarchies." The four layered models we have described can be—and frequently are—used simultaneously to represent different aspects of one system. Using the requirements and architecture methods, the required functional capabilities of a system are captured by the process model—an abstraction/detailing model; the required behavioral capabilities are captured by the control model—a controlling/controlled model; information structures in the system might include supertype/subtype relationships, captured in an entity-relationship model; and the physical

	Aggregation/ Decomposition	**Abstraction/ Detailing**	**Supertype/ Subtype**	**Controlling/ Controlled**
Aliases	Whole/part; container/content; composed-of; consists-of; assembly/ subassembly/ component	(Erroneously: abstraction/ decomposition)	Generalization/ specialization; class hierarchy; inheritance structure; "is-a" hierarchy	Is-boss-of/ is-supervised-by; uses hierarchy; client/server
Downward Usage	Decompose; dismantle	Add detail; specialize	Inherit; specialize	Control
Upward Usage	Aggregate; assemble	Make abstract; generalize	Set membership; "is-a"; generalize; categorize	Controlled by
Where Used, Roles	Architecture model; object orientation (aggregate objects); requirements and architecture dictionaries	Requirements model; Structured Analysis; nesting in statecharts	Architecture model when used with object orientation; entity-relationship-attribute modeling	Control model (of requirements model); Structured Design; system control structures (occurs within a layer as well as between layers)
Purpose of Usage	Physical packaging; information-hiding; defining scope and visibility	Coping with complexity; reducing complexity	Similar to abstraction/ detailing; in addition: inheritance of attributes, functions, behavior	Separation of concerns; creating noncyclic client/server structures; simplifying cooperation

Figure 3.18: Summary of Relationships in Layered Models.

structure is captured by the architecture model—an aggregation/decomposition model. Thus, to model a single system, not only can we use layered models of the same kind, as described in Section 2.1.3, we can also use layered models of different kinds. This allows us to represent different views of the system separately, but, when done as part of the requirements and architecture methods, the links between these views are carefully maintained.

This highlights the great flexibility of layered models. They are extraordinarily versatile, and allow us to represent just about any facet of systems and system development separately and at any desired level of detail, but with the links to the other facets also represented.

3.7 Model Framework Summary

Our modeling framework, shown in Figure 3.19—which is derived from Figure 3.7—now combines all the ideas described in this chapter. For most systems, especially larger ones, we exploit the idea that systems come in hierarchies. We have layers of specifications for the system, subsystems, sub-subsystem, and so on. But note that the flows between the layers go both ways—up and down. There is no sequence of development implied in this framework. We build groups of models as we discover subsystems in the hierarchy, and we do so in any order we want.

Forming subsystems is a difficult architectural or design decision, but exploitation of the information/material/energy classifications and of the generic subsystem categorization will guide us along the way. For software developers, we have many more guidelines in Chapter 6.

Each group of models separates the *what* from the *how:* We build separate requirements and architecture models. To help with the transition between *what* and *how* models, we build enhanced requirements models based on the architecture template. Thus, each group of models consists of three separate but related types: requirements, enhanced requirements, and architecture. Note that the arrows between the three models in the subsystems of Figure 3.19 go both ways—again, allowing these models to be developed in any sequence.

The information-hiding categories described in Section 3.6 are helpful for structuring all three models. They are generic enough to be applicable in many different application areas, yet precise enough to give a head start for partitioning. In many large applications, such subsystems have produced flexible, extendable, and maintainable systems. The case study in Part II demonstrates this idea with a specific example from a unique application domain.

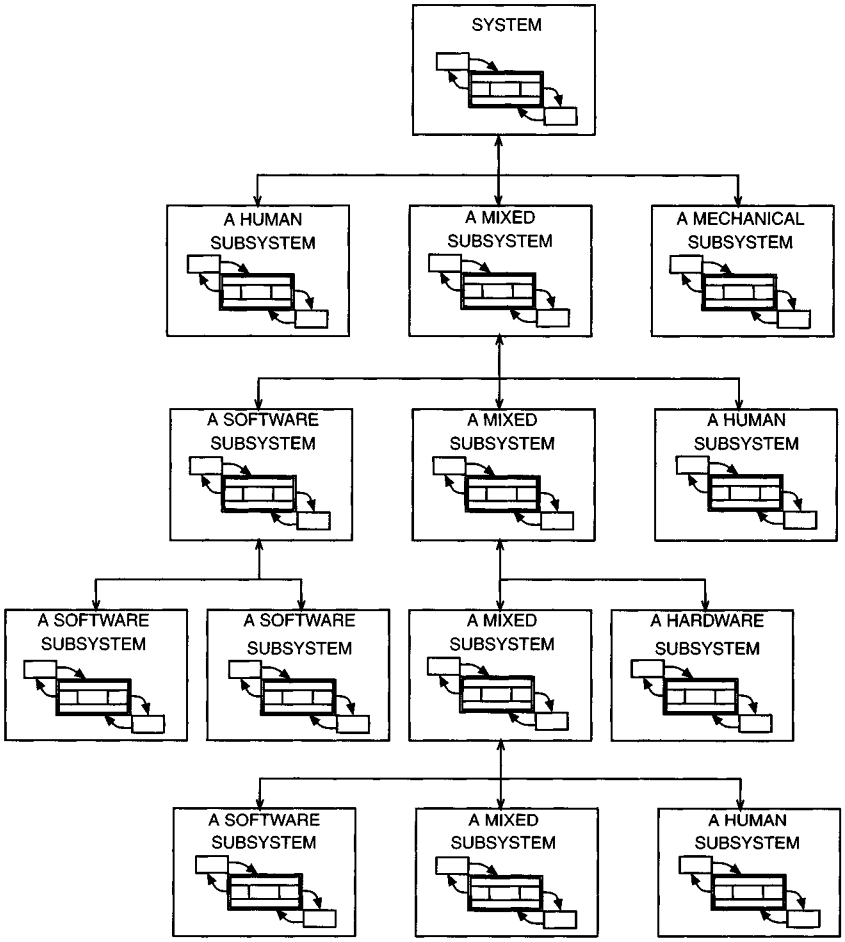


Figure 3.19: The Modeling Framework.

This page intentionally left blank

Index